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

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

Spin-based Approach

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

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

Spin-based Approach

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

```
volatile int done = 0;
1
2
    void *child(void *arg) {
3
        printf("child\n");
4
        done = 1;
5
        return NULL;
6
                              What's the problem of this approach?
    }
7
8
    int main(int argc, char *argv[]) {
9
        printf("parent: begin\n");
10
        pthread t c;
11
        Pthread_create(&c, NULL, child, NULL); // create child
12
        while (done == 0)
13
             ; // spin
14
        printf("parent: end\n");
15
        return 0;
16
17
    }
```

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;
}
```





<pre>void thread_exit() {</pre>		<pre>void thread_join() {</pre>	
<pre>Mutex_lock(&m);</pre>	// a	<pre>Mutex_lock(&m);</pre>	// x
<pre>Cond_signal(&c);</pre>	// b	Cond_wait(&c, &m);	// y
<pre>Mutex_unlock(&m);</pre>	// c	Mutex_unlock(&m);	// z
}		}	

Parent: x y		Z		
Child:	a b c			
<pre>void thread_exit() Mutex_lock(& Cond_signal(Mutex_unlock }</pre>	{ m); &c); (&m);	// a // b // c	<pre>void thread_join() { Mutex_lock(&m); Cond_wait(&c, &m); Mutex_unlock(&m); }</pre>	// x // y // z

Parent: x y		Z	
Child:	a b c	GOOD!	
<pre>void thread_exit() Mutex_lock(& Cond_signal Mutex_unlock }</pre>	{ wm); (&c); <(&m);	<pre>void thread_join() { // a Mutex_lock(&m); // b Cond_wait(&c, &m); // c Mutex_unlock(&m); }</pre>	// x // y // z

<pre>void thread_exit() {</pre>		<pre>void thread_join() {</pre>	
<pre>Mutex_lock(&m);</pre>	// a	<pre>Mutex_lock(&m);</pre>	// x
<pre>Cond_signal(&c);</pre>	// b	Cond_wait(&c, &m);	// y
<pre>Mutex_unlock(&m);</pre>	// c	Mutex_unlock(&m);	// z
}		}	

Parent:	Х	У
---------	---	---

Child: a b c

void	<pre>thread_exit() {</pre>			<pre>void thread_join() {</pre>
	<pre>Mutex_lock(&m);</pre>		а	Mutex_lock(&m);
	Cond_signal(&c);		b	Cond_wait(&c, &m);
	Mutex_unlock(&m);	11	С	Mutex unlock(&m);
}				}

// x

// y // z







Parent: w x y

Child: a b

void	thread_join() {			
	<pre>Mutex_lock(&m);</pre>		11	W
	if (done == 0)		11	х
	Cond_wait(&c,	&m);	11	У
	Mutex_unlock(&m);		11	Z
}				















Only one thread gets a signal

Condition Variable



Condition Variable







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



```
int done = 0;
1
    pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2
    pthread cond t c = PTHREAD COND INITIALIZER;
3
4
    void thr_exit() {
5
        Pthread_mutex_lock(&m);
6
        done = 1;
7
        Pthread_cond_signal(&c);
8
                                        Approach
        Pthread mutex unlock (&m);
9
10
11
    void *child(void *arg) {
12
        printf("child\n");
13
        thr_exit();
14
        return NULL;
15
16
17
    void thr_join() {
18
        Pthread mutex lock (&m);
19
        while (done == 0)
20
             Pthread_cond_wait(&c, &m);
21
        Pthread mutex unlock (&m);
22
23
24
    int main(int argc, char *argv[]) {
25
        printf("parent: begin\n");
26
        pthread_t p;
27
        Pthread_create(&p, NULL, child, NULL);
28
        thr join();
29
        printf("parent: end\n");
30
        return 0;
31
32
```

CV-based Parent-wait-for-child Approach

```
int done = 0;
1
    pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2
    pthread cond t c = PTHREAD COND INITIALIZER;
3
4
    void thr_exit() {
5
        Pthread_mutex_lock(&m);
6
                                       CV-based Parent-wait-for-child
        done = 1;
7
        Pthread_cond_signal(&c);
8
                                      Approach
        Pthread mutex unlock (&m);
9
10
11
    void *child(void *arg) {
12
                                      Good Rule of Thumb
        printf("child\n");
13
        thr_exit();
14
        return NULL;
15
                         Always do 1. wait and 2. signal while holding the lock
16
17
    void thr_join() {
18
        Pthread mutex lock (&m);
19
                                                   Why: To prevent lost signal
        while (done == 0)
20
            Pthread_cond_wait(&c, &m);
21
        Pthread mutex unlock (&m);
22
23
24
    int main(int argc, char *argv[]) {
25
        printf("parent: begin\n");
26
        pthread_t p;
27
        Pthread_create(&p, NULL, child, NULL);
28
        thr_join();
29
        printf("parent: end\n");
30
        return 0;
31
                                                                             35
32
```

Classical Problems of Synchronization

- Producer-consumer problem
 - CV-based version
- Readers-writers problem
- Dining-philosophers problem
```
Single CV and if statement
                                                           int buffer;
                                                       1
                                                           int count = 0; // initially, empty
                                                       2
                                                       3
                                                       4
                                                           void put(int value) {
                                                               assert(count == 0);
                                                       5
cond t cond;
                                                               count = 1;
                                                       6
mutex_t mutex;
                                                               buffer = value;
                                                       7
                                                       8
void *producer(void *arg) {
                                                       9
    int i;
                                                       10
                                                           int get() {
    for (i = 0; i < loops; i++) {
                                                               assert(count == 1);
                                                       11
        Pthread_mutex_lock(&mutex);
                                               // p1
                                                               count = 0;
                                                       12
        if (count == 1)
                                               // p2
                                                               return buffer;
                                                       13
            Pthread cond wait (&cond, &mutex); // p3
                                                       14
        put(i);
                                               // p4
                                               // p5
        Pthread cond signal (&cond);
                                                                Put and Get routines
        Pthread mutex unlock(&mutex);
                                               // p6
                                                                    Single buffer
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 = qet();
                                              // c4
        Pthread cond signal (&cond);
                                              // c5
        Pthread mutex unlock (&mutex);
                                               // c6
        printf("%d\n", tmp);
```

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
```

```
}
```

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

Put and Get routines Single buffer

What's the problem of this approach?

oid *consumer(void *arg) { C1 run	ning
int i;	0
for (i = 0; i < loops; i++) {	
<pre>Pthread_mutex_lock(&mutex);</pre>	// cl
if (count $== 0$)	// c2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	// c3
<pre>int tmp = get();</pre>	// c4
<pre>Pthread_cond_signal(&cond);</pre>	// c5
Pthread_mutex_unlock(&mutex);	// c6
<pre>printf("%d\n", tmp);</pre>	
}	

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

}

V

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



T_{c1}	State	T _{c2} State	State T_p	State Count	Comment
c1	Running	Ready	Ready	Ready 0	
c2	Running	Ready	Ready	Ready 0	
c3	Sleep	Ready	Ready	Ready 0	Nothing to get
	Sleep	Ready	Ready p1 R	unning 0	
	Sleep	Ready	Ready p2 R	unning 0	

}

V

<pre>void *consumer(void *arg) {</pre>		
int i;		
for (i = 0; i < loops; i++) {		
<pre>Pthread_mutex_lock(&mutex);</pre>	//	cl
if (count $== 0$)	11	с2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	11	с3
<pre>int tmp = get();</pre>	//	c4
<pre>Pthread_cond_signal(&cond);</pre>	11	c5
Pthread_mutex_unlock(&mutex);	11	C6
<pre>printf("%d\n", tmp);</pre>		
}		

void *producer(void *arg) { P running int i; for (i = 0; i < loops; i++) {</pre> Pthread_mutex_lock(&mutex); // p1 if (count == 1) // p2 Pthread_cond_wait(&cond, &mutex); // p3 put(i); 11 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

<pre>void *consumer(void *arg) {</pre>	
int i;	
for (i = 0; i < loops; i++) {	
<pre>Pthread_mutex_lock(&mutex);</pre>	// cl
if (count $== 0$)	// c2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	// c3
<pre>int tmp = get();</pre>	// c4
<pre>Pthread_cond_signal(&cond);</pre>	// c5
<pre>Pthread_mutex_unlock(&mutex);</pre>	// c6
<pre>printf("%d\n", tmp);</pre>	
A 1997 A 1977	



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	0 0
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken

oid *consumer(void *arg) {		
int i;		
for (i = 0; i < loops; i++) {		
<pre>Pthread_mutex_lock(&mutex);</pre>	// c	1
if (count $== 0$)	// c	2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	// c	3
<pre>int tmp = get();</pre>	// c	4
<pre>Pthread_cond_signal(&cond);</pre>	// c	5
<pre>Pthread_mutex_unlock(&mutex);</pre>	// c	6
<pre>printf("%d\n", tmp);</pre>		

void *producer(void *arg) { P running int i; for (i = 0; i < loops; i++) { Pthread_mutex_lock(&mutex); // p1 if (count == 1) p2 Pthread_cond_wait(&cond, &mutex); p3 11 put(i); p4 11 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	

}

V

<pre>void *consumer(void *arg) {</pre>	
int i;	
for (i = 0; i < loops; i++) {	
<pre>Pthread_mutex_lock(&mutex);</pre>	// cl
if $(count == 0)$	// c2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	// c3
<pre>int tmp = get();</pre>	// c4
<pre>Pthread_cond_signal(&cond);</pre>	// c5
Pthread_mutex_unlock(&mutex);	// c6
<pre>printf("%d\n", tmp);</pre>	
A 1997 A 1977	

void *producer(void *arg) { P running int i; for (i = 0; i < loops; i++) { Pthread_mutex_lock(&mutex); // p1 if (count == 1) p2 Pthread_cond_wait(&cond, &mutex); p3 11 put(i); p4 11 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	

void *consumer(void *arg) { C1 run	nable
for $(i = 0; i < loops; i++) $ {	
<pre>Pthread_mutex_lock(&mutex);</pre>	// cl
if $(count == 0)$	// c2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	// c3
<pre>int tmp = get();</pre>	// c4
<pre>Pthread_cond_signal(&cond);</pre>	// c5
<pre>Pthread_mutex_unlock(&mutex);</pre>	// c6
<pre>printf("%d\n", tmp);</pre>	
}	

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

}

oid *c	onsumer(void *arg) { C2 runr	ing	Į
int	i;		•
for	(i = 0; i < loops; i++) {		
Г	Pthread_mutex_lock(&mutex);	//	c1
	if (count == 0)	//	c2
	<pre>Pthread_cond_wait(&cond, &mutex);</pre>	11	c3
	<pre>int tmp = get();</pre>	11	c4
	Pthread_cond_signal(&cond);	11	c5
	Pthread_mutex_unlock(&mutex);	11	c6
	<pre>printf("%d\n", tmp);</pre>		
}			

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

}

V

void *consumer(void *arg) { C2 runr	ning
int i;	0
for (i = 0; i < loops; i++) {	
<pre>Pthread_mutex_lock(&mutex);</pre>	// c1
if (count $== 0$)	// c2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	// c3
<pre>int tmp = get();</pre>	// c4
<pre>Pthread_cond_signal(&cond);</pre>	// c5
<pre>Pthread_mutex_unlock(&mutex);</pre>	// c6
<pre>printf("%d\n", tmp);</pre>	
}	

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

void *co	onsumer(void *arg) { C2 runr	ning
int	i;	0
for	(i = 0; i < loops; i++) {	
	<pre>Pthread_mutex_lock(&mutex);</pre>	// c1
	if (count $== 0$)	// c2
	<pre>Pthread_cond_wait(&cond, &mutex);</pre>	// c3
	<pre>int tmp = get();</pre>	// c4
	Pthread_cond_signal(&cond);	// c5
	Pthread_mutex_unlock(&mutex);	// c6
	<pre>printf("%d\n", tmp);</pre>	
}		

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

<pre>void *consumer(void *arg) { C2 run</pre>	ning
int i;	0
for (i = 0; i < loops; i++) {	
<pre>Pthread_mutex_lock(&mutex);</pre>	// c1
if (count $== 0$)	// c2
Pthread_cond_wait(&cond, &mutex);	; // c3
<pre>int tmp = get();</pre>	// c4
Pthread_cond_signal(&cond);	// c5
<pre>Pthread_mutex_unlock(&mutex);</pre>	// c6
<pre>printf("%d\n", tmp);</pre>	
}	

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	-

void *co	onsumer(void *arg) { C1 runr	ning	
int	i;	0	
for	(i = 0; i < loops; i++) {		
	Pthread_mutex_lock(&mutex);	11	c1
	if (count $== 0$)	11	c2
	<pre>Pthread_cond_wait(&cond, &mutex);</pre>	11	c3
	<pre>int tmp = get();</pre>	//	c4
	Pthread_cond_signal(&cond);	11	c5
	Pthread_mutex_unlock(&mutex);	11	c6
	<pre>printf("%d\n", tmp);</pre>		
}			

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

}

Y. Cheng

}

GMU CS571 Spring 2020

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

```
cond t cond;
1
    mutex_t mutex;
2
3
                                                  Single CV and while
    void *producer(void *arg) {
4
        int i;
5
        for (i = 0; i < loops; i++) {
6
             Pthread mutex lock (&mutex);
                                                      // p1
7
             while (count == 1)
                                                      // p2
8
                 Pthread_cond_wait(&cond, &mutex);
                                                      // p3
9
             put(i);
                                                      // p4
10
             Pthread_cond_signal(&cond);
                                                      // p5
11
            Pthread mutex unlock (&mutex);
                                                      // p6
12
         }
13
14
    }
                                                   What's the problem of this
15
    void *consumer(void *arg) {
16
                                                   approach?
        int i;
17
18
        for (i = 0; i < loops; i++)  {
             Pthread_mutex_lock(&mutex);
                                                      // c1
19
             while (count == 0)
                                                      // c2
20
                 Pthread_cond_wait(&cond, &mutex);
                                                      // c3
21
             int tmp = qet();
                                                      // c4
22
             Pthread_cond_signal(&cond);
                                                      // c5
23
             Pthread mutex unlock (&mutex);
                                                      // c6
24
             printf("%d\n", tmp);
25
         }
26
                                                                          52
27
```



voi	<pre>void *consumer(void *arg) { C2 running int i;</pre>						void *produc int i;	er(void *a	rg) {		
	foi	r (i = 0)	; i < loops;	i++) {			for (i =	= 0; i < lo	ops; i++) {		
		Pthrea	d_mutex_lock	(&mutex)	;	// c1	Pthr	ead_mutex_	lock(&mutex);	// p1	
		while	(count == 0)			// c2	whil	e (count =	= 1)	// p2	
		Pt	hread_cond_wa	ait(&con	d, &mutex);	// c3	<pre>Pthread_cond_wait(&cond, &mutex);</pre>				
		int tm	p = get();			// c4	put (i);		// p4	
Pthread_cond_signal(&cond); // c5							Pthr	read_cond_s	ignal(&cond);	// p5	
Pthread_mutex_unlock(&mutex); // c6						Pthread_mutex_unlock(&mutex); // p6					
<pre>printf("%d\n", tmp);</pre>							}				
	}						}				
}		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		

```
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
                          // c4
       int tmp = get();
       Pthread_cond_signal(&cond);
                                          // c5
       Pthread_mutex_unlock(&mutex);
                                           // c6
       printf("%d\n", tmp);
    }
```

<pre>void *producer(void *arg) { int i; Prunr</pre>	ing	
for (i = 0; i < loops; i++) {		
<pre>Pthread_mutex_lock(&mutex);</pre>	11	p1
while (count == 1)	11	p2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	11	pЗ
put(i);	//	p4
Pthread_cond_signal(&cond);	11	p5
Pthread_mutex_unlock(&mutex);	11	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	0 0
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	0
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now 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
                          // c4
       int tmp = get();
       Pthread_cond_signal(&cond);
                                           // c5
       Pthread_mutex_unlock(&mutex);
                                           // c6
       printf("%d\n", tmp);
    }
```

<pre>void *producer(void *arg) {</pre>	ning	
for (i = 0; i < loops; i++) {		
<pre>Pthread_mutex_lock(&mutex);</pre>	11	p1
while (count == 1)	11	p2
Pthread_cond_wait(&cond, &mutex);	: //	pЗ
put(i);	11	p4
<pre>Pthread_cond_signal(&cond);</pre>		р5
<pre>Pthread_mutex_unlock(&mutex);</pre>	//	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	0 0
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	0 0
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken

void *p int	<pre>producer(void *arg) { P sleep i;</pre>	ing	
for	(i = 0; i < loops; i++) {		
	<pre>Pthread_mutex_lock(&mutex);</pre>	11	p1
	while (count == 1)	11	p2
	<pre>Pthread_cond_wait(&cond, &mutex);</pre>	11	p3
	put(i);	11	p4
	Pthread_cond_signal(&cond);	11	p5
	Pthread_mutex_unlock(&mutex);	11	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	0 0
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	0 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)

vo	<pre>vid *consumer(void *arg) { C1 running</pre>						<pre>void *producer(void *arg) { int i.</pre>			
	for	$ \perp i = 0 $	· i < loopa.	; + +) (for $(i = 0; i < loops; i++) $			
	101	D + bros	d mutex lock	$(\mathfrak{L} T T T T T T T T$		11 01	Pthread mutex lock (&mutex):			
		while	(count == 0)	(anuces)	/	$// c^2$	while $(count == 1)$			
		Pt	hread cond wa	ait (&con	d. &mutex):	$// c_{3}$		Pthread_co	nd_wait(&cond, &mutex);	
		int tm	p = qet();		a, anacon,,	put (i);			
		Pthrea	d_cond_signal	L(&cond)	;	// c5	Pthr	ead_cond_s	ignal(&cond);	
		Pthrea	d_mutex_unloc	ck (&mute	x);	// c6	Pthr	ead_mutex_	unlock(&mutex);	
		printf	("%d\n", tmp)	;			}			
	}						}			
}		T_{c1}	State	T_{c2}	State	$\mid T_p$	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	0 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	

// p1

// p2

// p3

// p4 // p5

// p6

voi	d *c' int for	onsumer i; (i = 0, Pthread while Pthread Pthread Pthread printf	<pre>(void *arg) { ; i < loops; d_mutex_lock (count == 0) nread_cond_wa p = get(); d_cond_signal d_mutex_unloc ("%d\n", tmp)</pre>	i++) { (&mutex) ait(&cond) ck(&mute ;	<pre>C1 runnir ; d, &mutex); ; x);</pre>	// c1 // c2 // c3 // c4 // c5 // c6	<pre>void *produce int i; for (i = Pthrewhile while put(Pthrew } }</pre>	er(void *ar 0; i < loc ead_mutex_1 e (count == Pthread_con i); ead_cond_s: ead_mutex_u	rg) ops; lock = 1) nd w igna unlc
}	J	Tal	State	Taz	State	T_{n}	State	Count	1
	-	c1	Running	-02	Ready	- p	Ready	0	
		c2	Running		Ready		Ready	0	
		c3	Sleep		Ready		Ready	0	
			Sleep	c1	Running		Ready	0	
			Sleep	c2	Running		Ready	0	
			Sleep	c3	Sleep		Ready	0	
			Sleep		Sleep	p1	Running	0	
			Sleep		Sleep	p2	Running	0	
			Sleep		Sleep	p4	Running	1	
			Ready		Sleep	p5	Running	1	
			Ready		Sleep	p6	Running	1	
			Ready		Sleep	p1	Running	1	
			Ready		Sleep	p2	Running	1	
			Ready		Sleep	p3	Sleep	1]
		c2	Running		Sleep		Sleep	1	R
		c4	Running		Sleep		Sleep	0	

{ i++) { // p1 // p2 // p3 // p4 // p5 // p6 k(&mutex);) wait(&cond, &mutex); al(&cond); ock(&mutex); Commont

I_{c1}	State	1_{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

void	*C0	onsumer(void *arg)	{	C1 runni	ng	v
	int	1;		`			
	for	(i = 0;					
		Pthread	// c1				
		while (// c2				
		Pth	read_cond_w	ait(&cond,	&mutex);	// c3	
		int tmp	= get();			// c4	
		Pthread	_cond_signa	l(&cond);		// c5	
		Pthread	_mutex_unlo	ck(&mutex)	;	// c6	
		printf("%d\n", tmp);			
	}	÷					}
}		T_{c1}	State	T _{c2}	State	$ T_p $	

void *producer(void *arg) { int i; for (i = 0; i < loops; i++) {</pre> Pthread_mutex_lock(&mutex); // p1 while (count == 1)
 Pthread_cond_wait(&cond, &mutex); // p2 // 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}

<pre>void *consumer(void *arg) {</pre>						<pre>void *producer(void *arg) { int i;</pre>			
fo	r (i = 0 Pthrea while	; i < loops; d_mutex_lock (count == 0)	i++) { (&mutex)	;	<pre>for (i = 0; i < loops; i++) { Pthread_mutex_lock(&mutex); while (count == 1)</pre>				
<pre>Pthread_cond_wait(&cond, &mutex); //</pre>					// c3	Pthread cond wait (& cond, & mi			
<pre>int tmp = get(); //</pre>				// c4	put(1); Pthroad cond signal(scond):				
Pthread_cond_signal(&cond); /				1/ 05	Pthread mutex unlock (&mutex):				
<pre>Printead_mutex_unlock(&mutex); // printf("%d\n"tmp):</pre>					// 00	}			
}	Fringr	(•••(11) •••(12)	/			}			
}	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	1	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	1 02	
	c1	Running		Ready		Sleep	0		
	c2	Running		Ready		Sleep	0		
	c3	Sleep		Ready		Sleep	0	Nothing to get	

// p1

// p2

// p3 // p4 // p5

// p6

voi	d *c int	onsumer i;	(void *arg)	{	C2 runnin	g	void *produce int i;	er(void *a:	rg) {	
	for	(i = 0	; i < loops;	i++) {			for $(i = 0; i < loops; i++) $ {			
		Pthrea	d_mutex_lock	(&mutex)	;	// c1	Ptnre while	ead_mutex e (count ==	= 1)	// pl
	Ŧ	wnile Pt	(Count == 0)	ait(&con	d. &mutex):	$// c_2$		Pthread_co	nd_wait(&cond, &mutex);	// p3
		int tm	p = get();	are (acon	ay anaconyy	// c4	put (i);		// p4
	Pthread_cond_signal(&cond); // c5						Pthread_cond_signal(&cond);			
	Pthread_mutex_unlock(&mutex); // c6						Pthread_mutex_unlock(&mutex); // p			
	ı	printi	("%d\n", tmp);			}			
}	ſ	T_{c1}	State	T _{c2}	State	T_p	State	Count	Comment	
		c1	Running		Ready	P	Ready	0		
		c2	Running		Ready		Ready	0		
		c3	Sleep		Ready		Ready	0	Nothing to get	
			Sleep	c1	Running		Ready	0	0 0	
			Sleep	c2	Running		Ready	0		
			Sleep	c3	Sleep		Ready	0	Nothing to get	
			Sleep		Sleep	p1	Running	0	0 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	1. A	
		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	

```
void *consumer(void *arg) {
                                      C2 sleeping
    int i;
    for (i = 0; i < loops; i++) {</pre>
        Pthread_mutex_lock(&mutex);
                                                // c1
        while (count == 0)
                                                // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();
                                                // c4
        Pthread_cond_signal(&cond);
                                               // c5
                                                // c6
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
```

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

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

27

```
cond_t Cempty, fill;
1
    mutex_t mutex;
2
3
                                                  Two CVs and while
    void *producer(void *arg) {
4
        int i;
5
        for (i = 0; i < loops; i++) {
6
             Pthread_mutex_lock(&mutex);
7
             while (count == 1)
8
                 Pthread_cond_wait(&empty, &mutex);
9
             put(i);
10
             Pthread_cond_signal(&fill);
11
             Pthread mutex unlock (&mutex);
12
13
         }
                                              Using two CVs to distinguish two
14
    }
                                           types of threads; in order to properly
15
                                            signal which thread should wake up
    void *consumer(void *arg) {
16
        int i;
17
                                                      Producer waits on empty
        for (i = 0; i < loops; i++)  {
18
                                                        Consumer waits on full
             Pthread_mutex_lock(&mutex);
19
             while (count == 0)
20
                 Pthread_cond_wait(&fill, &mutex);
21
             int tmp = qet();
22
             Pthread_cond_signal(&empty);
23
             Pthread_mutex_unlock(&mutex);
24
             printf("%d\n", tmp);
25
26
                                                                         65
```

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

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

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

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

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

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

```
typedef struct _rwlock_t {
1
                        // binary semaphore (basic lock)
      sem t lock;
2
      sem t writelock; // used to allow ONE writer or MANY readers
3
                        // count of readers reading in critical section
      int
            readers;
4
    } rwlock t;
5
6
                                                     Initially, # readers is 0
    void rwlock_init(rwlock_t *rw) {
7
      rw->readers = 0;
8
                                                     binary sem lock set to 1
      sem_init(&rw->lock, 0, 1);
9
                                                    writelock set to 1
      sem_init(&rw->writelock, 0, 1);
10
11
    }
12
    void rwlock_acquire_readlock(rwlock_t *rw) {
13
      sem wait(&rw->lock);
14
      rw->readers++;
15
      if (rw->readers == 1)
16
        sem wait(&rw->writelock); // first reader acquires writelock
17
      sem_post(&rw->lock);
18
19
    }
                                                                                 Writer cannot
20
    void rwlock_release_readlock(rwlock_t *rw) {
                                                                                 be in CS when
21
      sem wait(&rw->lock);
22
                                                                                 readers are!
      rw->readers--;
23
      if (rw \rightarrow readers == 0)
24
        sem_post(&rw->writelock); // last reader releases writelock
25
      sem_post(&rw->lock);
26
27
    }
28
    void rwlock_acquire_writelock(rwlock_t *rw) {
29
      sem wait(&rw->writelock);
30
    }
31
32
    void rwlock release writelock(rwlock t *rw) {
33
      sem_post(&rw->writelock);
34
                                                                                             73
35
```

Readers-Writers Problem: Writer Thread

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

Philosopher 0

Philosopher 2

Philosopher 4 Philosopher 3

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 GMU CS571 Spring

Shared data

```
sem t forks[5];
```

Initially all semaphore values are 1

Dining-Philosophers Problem

• The basic loop of a philosopher



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

```
void getforks() {
1
2
      sem_wait(forks[left(p)]);
      sem_wait(forks[right(p)]);
3
    }
4
5
                                      Is this solution correct?
    void putforks() {
6
      sem_post(forks[left(p)]);
7
      sem_post(forks[right(p)]);
8
9
 Y. Cheng
                         GMU CS571 Spring 2020
```

Thread 0

Interleaving

Thread 1

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

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

Thread 0

Interleaving

Thread 1

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

sem wait(fork[0])

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

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

Thread 0

Interleaving

Thread 1

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

sem_wait(fork[0])

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

sem wait(fork[0])

sem wait(fork[1])

Thread 0

Interleaving

Thread 1

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

sem_wait(fork[0])

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

sem_wait(fork[0])
 wait...

sem wait(fork[1])

sem_wait(fork[1])

Thread 0

Interleaving

Thread 1

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

sem_wait(fork[0])

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

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

sem wait(fork[1])

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

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

Thread 0	Interleaving	Thread 1
<pre>sem_wait(fork[0]) sem_wait(fork[1]) sem_signal(fork[0]) sem_signal(fork[1])</pre>	<pre>sem_wait(fork[0])</pre>	<pre>sem_wait(fork[1]) sem_wait(fork[0]) sem_signal(fork[1]) sem_signal(fork[0])</pre>
	<pre>sem_wait(fork[1])</pre>	
	<pre>sem_wait(fork[0]) wait</pre>	

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

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

- 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])

- 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

- 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 cat 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

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