CS 471 Operating Systems

Yue Cheng

George Mason University Spring 2019

Review: Condition Variables Worksheet

Condition Variables

- CV: Queue of sleeping threads
- Threads add themselves to the queue with wait
- Threads wake up threads on the queue with signal

Condition Variables

- o cond_wait(cond_t *cv, mutex_t *lock)
 - assume the lock is held when cond_wait() is called
 - puts caller to sleep + release the lock (atomically)
 - when awaken, reacquires lock before returning
- o cond_signal(cond_t *cv)
 - wake a single waiting thread (if >= 1 thread is waiting)
 - if there is no waiting thread, just return, doing nothing

<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	<pre>Mutex_unlock(&m);</pre>	// z
}		}	

Parent: x y		Z	2	
Child:	a b c			
<pre>void thread_exit() Mutex_lock(Cond_signal Mutex_unloc }</pre>	{ &m); (&c); k(&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(</pre>) { &m); (&c); :k(&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	<pre>Mutex_unlock(&m);</pre>	// z
}		}	

Parent: x y

Child: a b c

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

// x

// y // z

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

Parent: w x y

Child: a b





Good Rule of Thumb When Using CV

Always do wait and signal while holding the lock

Good Rule of Thumb When Using CV

Always do wait and signal while holding the lock

Why? To prevent lost signals.

Classical Problems of Synchronization

- Producer-Consumer Problem
 - CV-based version
- Readers-Writers Problem
- Dining-Philosophers Problem

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
                                               // p5
        Pthread cond signal (&cond);
        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
    int get() {
10
        assert(count == 1);
11
        count = 0;
12
        return buffer;
13
14
         Put and Get routines
```

```
Single buffer
```

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?

void *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	
<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

void *consumer(void *arg) {		
int i;		
for (i = 0; i < loops; i++) {		
<pre>Pthread_mutex_lock(&mutex);</pre>	// c1	
if (count $== 0$)	// c2	1
<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	

void *consumer(void *arg) {	
int i;	
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

void *consumer(void *arg) {		
int i;		
for (i = 0; i < loops; i++) {		
<pre>Pthread_mutex_lock(&mutex);</pre>	// (c1
if (count $== 0$)	11 0	c2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	11 0	c3
<pre>int tmp = get();</pre>	// ‹	c4
<pre>Pthread_cond_signal(&cond);</pre>	11 0	c5
<pre>Pthread_mutex_unlock(&mutex);</pre>	11 0	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
			-	_	-		

void

void *consumer(void *arg) {	
int i;	
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>	

*p:	roducer(void *arg) {		
int	i;		
Eor	(i = 0; i < loops; i++) {		
	<pre>Pthread_mutex_lock(&mutex);</pre>	11	p1
	if (count == 1)	//	p2
	<pre>Pthread_cond_wait(&cond, &mutex);</pre>	11	ъЗ
			T
	put(i);	11	p4
	<pre>put(i); Pthread_cond_signal(&cond);</pre>	11	p4 p5
	<pre>put(i); Pthread_cond_signal(&cond); Pthread_mutex_unlock(&mutex);</pre>	 	p4 p5 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	1.11.1 (M.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1

void

void *consumer(void *arg) {		
int i;		
for (i = 0; i < loops; i++) {		
<pre>Pthread_mutex_lock(&mutex);</pre>	//	cl
if (count $== 0$)	11	c2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	11	с3
<pre>int tmp = get();</pre>	//	c4
<pre>Pthread_cond_signal(&cond);</pre>	11	c5
<pre>Pthread_mutex_unlock(&mutex);</pre>	11	c6
<pre>printf("%d\n", tmp);</pre>		

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

1

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

void *co	onsumer(void *arg) {		
int	i;		
for	(i = 0; i < loops; i++) {		
	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
	<pre>Pthread_cond_signal(&cond);</pre>	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

void *consumer(void *arg) {		
int i;		
for (i = 0; i < loops; i++) {		
<pre>Pthread_mutex_lock(&mutex);</pre>	11	c1
if (count $== 0$)	11	c2
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	11	с3
<pre>int tmp = get();</pre>	//	c4
<pre>Pthread_cond_signal(&cond);</pre>	//	c5
<pre>Pthread_mutex_unlock(&mutex);</pre>	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
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	and grabs data

void *co	onsumer(void *arg) {		
int	i;		
for	(i = 0; i < loops; i++) {		
	<pre>Pthread_mutex_lock(&mutex);</pre>	11	c1
	if $(count == 0)$	11	c2
	<pre>Pthread_cond_wait(&cond, &mutex);</pre>	11	c3
_	<pre>int tmp = get();</pre>	11	c4
C	<pre>Pthread_cond_signal(&cond);</pre>	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
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	and grabs data
	Ready	c5	Running		Ready	0	T_p awoken

void *co	onsumer(void *arg) {		
int	i;		
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>	11	c4
_	<pre>Pthread_cond_signal(&cond);</pre>	11	c5
	<pre>Pthread_mutex_unlock(&mutex);</pre>	11	с6
	<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	<u>^</u>

}

void *consumer(void *arg) {	
int i;	
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>	
}	

<pre>void *producer(void *arg) {</pre>		
int i;		
for (i = 0; i < loops; i++) {		
<pre>Pthread_mutex_lock(&mutex);</pre>	11	p1
if (count == 1)	11	p2
Pthread_cond_wait(&cond, &mutex)	; //	р3
put(i);	//	p4
<pre>Pthread_cond_signal(&cond);</pre>	11	p5
<pre>Pthread_mutex_unlock(&mutex);</pre>	11	p6
}		

\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	0 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

```
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
                                                      // p5
            Pthread_cond_signal(&cond);
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) {
                                                  approach?
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
                                                                           33
27
```

<pre>void *consumer(void *arg) { int i; for (i = 0; i < loops; i++) {</pre>							rg) { pps; i++) {					
	TOT	Pthrea	d_mutex_lock	(&mutex);		// c1		Pthr	ead_mutex_	lock (&mutex);	11	p1
		while	(count == 0)			// c2		whil	e (count ==	= 1)	11	p2
	1	Pt	hread_cond_wa	ait(&cond,	<pre>&mutex);</pre>	// c3			Pthread_com	nd_wait(&cond, &mutex);	11	pЗ
		int tm	p = get();			// c4		put (i);		11	p4
Pthread cond signal (& cond); // c5 Pthread_cond_signal (& cond);							ignal(&cond);	11	p5			
		Pthrea	d mutex unlo	ck(&mutex)	;	// c6		Pthr	ead_mutex_	unlock(&mutex);	11	p6
		printf	("%d\n", tmp));				}				
	}	I	· · · · · · · · · · · · · · · · · · ·				}					
}		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		

void	l *c int	onsumer i;	(void *arg) {		C2		void *producint i;	cer
	for	(i = 0	; i < loops;	i++) {			for (i :	= 0
		Pthrea	d_mutex_lock	(&mutex)	;	// c1	Pth	rea
		while	(count == 0)			// c2	whi	le
	Г	Pt	hread_cond_wa	ait(&con	d, &mutex);	// c3		Pt
		int tm	p = get();			// c4	put	(i)
		Pthrea	d_cond_signal	(&cond)	;	// c5	Pthrea	
		Pthrea	d mutex unloc	ck (&mute	x);	// c6	Pth	rea
		printf	("%d\n", tmp)	;			}	
	}	I	, , , , , , , , , , , , , , , , , , , ,	,			}	
}		T_{c1}	State	T_{c2}	State	$\mid T_p$	State	
	-	c1	Running		Ready		Ready	
		c2	Running		Ready		Ready	
		c3	Sleep		Ready		Ready	
			Sleep	c1	Running		Ready	
			C1 1	•	D . U			

```
(void *arg) {
; i < loops; i++) {
                                // p1
id_mutex_lock(&mutex);
(count == 1)
                                // p2
chread_cond_wait(&cond, &mutex); // p3
                                // p4
;
d_cond_signal(&cond);
                                // p5
                                // p6
d_mutex_unlock(&mutex);
```

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++) {</pre>
                                               // c1
        Pthread_mutex_lock(&mutex);
                                               // c2
        while (count == 0)
      Pthread_cond_wait(&cond, &mutex); // c3
                                              // c4
        int tmp = get();
        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++) $ {		
<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
<pre>Pthread_cond_signal(&cond);</pre>	11	p5
<pre>Pthread_mutex_unlock(&mutex);</pre>	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
```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {</pre>
                                               // c1
        Pthread_mutex_lock(&mutex);
                                               // c2
        while (count == 0)
      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; for (i = 0; i < loops; i++) {</pre>		
<pre>Pthread_mutex_lock(&mutex);</pre>	11	p1
while (count == 1)	11	p2
Pthread_cond_wait(&cond, &mutex)	; 11	p3
put(i);	11	p4
<pre>Pthread_cond_signal(&cond);</pre>	//	рb
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	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

}

}

<pre>void *producer(void *arg)</pre>	{	
int i;		
for $(i = 0; i < loops)$; i++) {	
Pthread_mutex_lock	k(&mutex); //	p1
while (count == 1)) //	p2
Pthread cond w	wait(&cond, &mutex); //	ъЗ
put(i);	//	p4
Pthread_cond_signa	al(&cond); //	p5
Pthread_mutex_unlo	ock(&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	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)

}

void	*co int	onsumer i;	(void *arg) {		C1		vo
	for	(i = 0	; i < loops;	i++) {			
		Pthrea	d_mutex_lock	(&mutex)	;	// c1	
	Г	while	(count == 0)			// cZ	
		Pt	hread_cond_wa	iit (&cond	d, &mutex);	// c3	
		int tm	// c4				
		Pthrea	// c5				
		Pthrea	d_mutex_unloc	k (&mute:	x);	// c6	
		printf	("%d\n", tmp)	;			
	}						}
}		T_{c1}	State	T_{c2}	State	$ T_p $	
		c1	Running		Ready		
		c2	Running		Ready		
		c3	Sleep		Ready		
			Sleep	c1	Running		

id *producer(void *arg) {		
int i;		
for $(i = 0; i < loops; i++) $ {		
<pre>Pthread_mutex_lock(&mutex);</pre>	11	p1
while (count == 1)		p2
Pthread cond wait (&cond, &mutex)	; //	pЗ
put(i);	11	p4
<pre>Pthread_cond_signal(&cond);</pre>	11	p5
<pre>Pthread_mutex_unlock(&mutex);</pre>	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)
c2	Running		Sleep		Sleep	1	Recheck condition

voi	d *c int for	onsumer i; (i = 0 Pthrea while Pt int tm Pthrea Pthrea printf	<pre>(void *arg) { ; i < loops; d_mutex_lock (count == 0) hread_cond_wa p = get(); d_cond_signal d_mutex_unloc ("%d\n", tmp)</pre>	i++) { (&mutex) ait(&con (&cond) ck(&mute ;	C1 ; d, &mutex); ; x);	// c1 // c2 // c3 // c4 // c5 // c6	<pre>void *product int i; for (i = Pthrewhild put(Pthrewhild } }</pre>	er(void *a: 0; i < loo ead_mutex_: e (count == Pthread_con i); ead_cond_s: ead_mutex_u	rg) ops loc = 1 nd igr unl
}		T_{c1}	State	T_{c2}	State	$ T_p $	State	Count	
		c1	Running		Ready		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	
		c4	Running		Sleep		Sleep	0	

{ s; i++) { ck(&mutex); // p1 1) // p2 wait(&cond, &mutex); // p3 // p4 // p5 // p6 nal(&cond); lock(&mutex);

Γ_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c 1	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
c4	Running		Sleep		Sleep	0	T_{c1} grabs data

void *c int for	onsumer i; (i = 0 Pthrea while Pt int tm Pthrea Pthrea printf	<pre>(void *arg) { ; i < loops; .d_mutex_lock (count == 0) hread_cond_wa p = get(); .d_cond_signal .d_mutex_unloc ("%d\n", tmp)</pre>	i++) { (&mutex) ait(&con (&cond) ck(&mute ;	C1 ; d, &mutex); ; x);	// c1 // c2 // c3 // c4 // c5 // c6	<pre>void *produ int i; for (i Pth whi put Pth Pth }</pre>
}	T_{c1}	State	T_{c2}	State	$\mid T_p$	State
	c1 c2 c3	Running Running Sleep Sleep Sleep Sleep Sleep Sleep Ready Ready	c1 c2 c3	Ready Ready Ready Running Running Sleep Sleep Sleep Sleep Sleep Sleep	p1 p2 p4 p5	Ready Ready Ready Ready Ready Ready Running Running Running 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	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}

void	d *c int for	onsumer i; (i = 0 Pthrea while Pt int tm Pthrea Pthrea printf	<pre>(void *arg) { ; i < loops; d_mutex_lock (count == 0) hread_cond_wa p = get(); d_cond_signal d_mutex_unloc ("%d\n", tmp)</pre>	// c1 // c2 // c3 // c4 // c5 // c6	<pre>void *product int i; for (i = Pthrewhile while put() Pthrew } }</pre>	er(void *a 0; i < lo ead_mutex_ e (count = Pthread co i); ead_cond_s ead_mutex_		
}		T_{c1}	State	T_{c2}	State	T_p	State	Count
	-	c1	Running		Ready		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
		c4	Running		Sleep		Sleep	0
		c5	Running		Ready		Sleep	0
		c6	Running		Ready		Sleep	0
		c 1	Running		Ready		Sleep	0
		c2	Running		Ready		Sleep	0
		c3	Sleep		Ready		Sleep	0

rg) { ops; i++) { lock(&mutex); // p1 = 1) // p2 ond wait(&cond, &mutex); // p3 // p4 signal(&cond); // p5 // p6 unlock(&mutex);

Comment

Nothing to get

Nothing to get

Buffer now full T_{c1} awoken

Must sleep (full) Recheck condition

 T_{c1} grabs data Oops! Woke T_{c2}

Nothing to get

<pre>void *consumer(void *arg) { int i; for (i = 0; i < loops; i++) { Pthread_mutex_lock(&mutex);</pre>						<pre>void *producer(void *arg) int i; for (i = 0; i < loops; Pthread_mutex_lock while (count == 1) Pthread_cond_wa put(i); Pthread_cond_signal Pthread_mutex_unloc } </pre>			
}	T_{c1}	State	T_{c2}	State	$\mid T_p$	State	Count		
	c1	Running		Ready		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	I	
		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	N	
	c2	Running		Sleep		Sleep	1	Re	
	c4	Running		Sleep		Sleep	0		
	c5	Running		Ready		Sleep	0	C	
	c6	Running		Ready		Sleep	0		
	c1	Running		Ready		Sleep	0		
	c2	Running		Ready		Sleep	0		
	c3	Sleep		Ready		Sleep	0]	
		Sleep	c2	Running		Sleep	0		
		Sleep	c3	Sleep		Sleep	0	Ex	

1 1 i++) { // p1 (&mutex); // p2 ait(&cond, &mutex); // p3 // p4 // p5 // p6 l(&cond); ck(&mutex);

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	
c 1	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

T_{c1}	State	T_{c2}	State	$\mid T_p$	State	Count	Comment
c 1	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
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Running		Ready		Sleep	0	
c 1	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

```
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
```

CV-based Producer-Consumer Implementation 3

```
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
                                                                          46
26
```

27

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

```
tvpedef 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
      int
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
      sem t lock;
                        // binary semaphore (basic 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
      sem t lock;
                        // binary semaphore (basic 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
7
    void rwlock_init(rwlock_t *rw) {
      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
      sem t lock;
                        // binary semaphore (basic 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->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
      sem t lock;
                        // binary semaphore (basic 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
    }
                                                                                 Writer cannot be
20
    void rwlock_release_readlock(rwlock_t *rw) {
21
                                                                                 in CS when
      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
                                                                                               54
      sem_post(&rw->writelock);
34
```

35

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?

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



Shared data

```
sem t forks[5];
Initially all semaphore values are 1
```

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 59 the other

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
```

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

```
sem_wait(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])
 wait...

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

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

Simplest E	Example of A	Deadlock
Q: Would the pr exactly the s	revious 5DP implem ame deadlock as sh	entation cause own 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>	
	<pre>sem_wait(fork[1]) wait</pre>	

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] are 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?
Dijkstra's Solution: Break the Circular Wait Condition

- Change how forks are acquired by at least one of the philosophers
- \circ 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