CS 471 Operating Systems

Yue Cheng

George Mason University Fall 2019

Announcements

- HW1 posted yesterday on BB
 - Due end of Friday 09/20
- OS/161 PA1 (Synchronization) will be posted on BB this Thursday 09/19
 - Due end of day 10/18

Review: Condition Variables

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
Cond_signal(&c);	// b	Cond_wait(&c, &m);	// y
<pre>Mutex_unlock(&m);</pre>	// c	<pre>Mutex_unlock(&m);</pre>	// z
}		}	

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

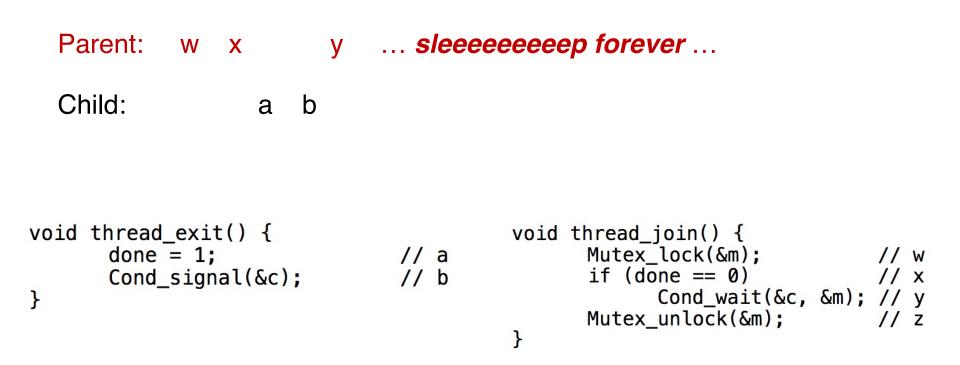
<pre>void thread_exit() {</pre>		VO
<pre>Mutex_lock(&m);</pre>	// a	
Cond_signal(&c);	// b	
<pre>Mutex_unlock(&m);</pre>	// c	
}		}

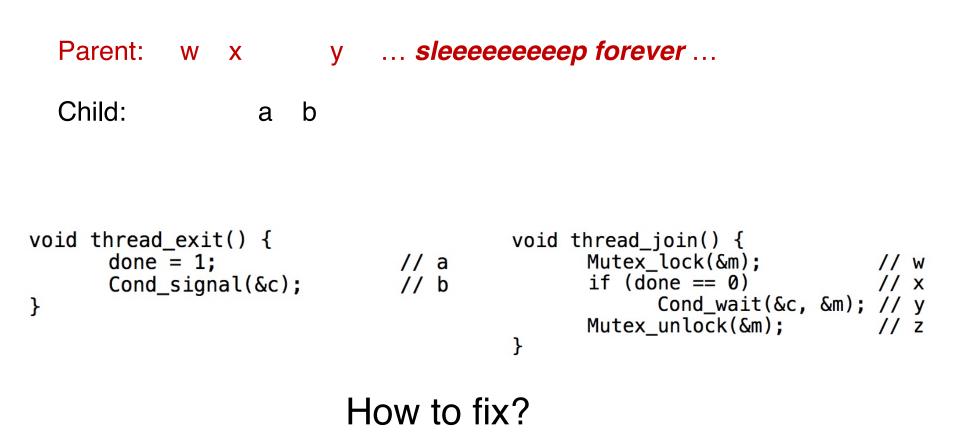
void	thread_join() {	
	<pre>Mutex_lock(&m);</pre>	// x
	<pre>Cond_wait(&c, &m);</pre>	// y
	<pre>Mutex_unlock(&m);</pre>	// z
}	_	

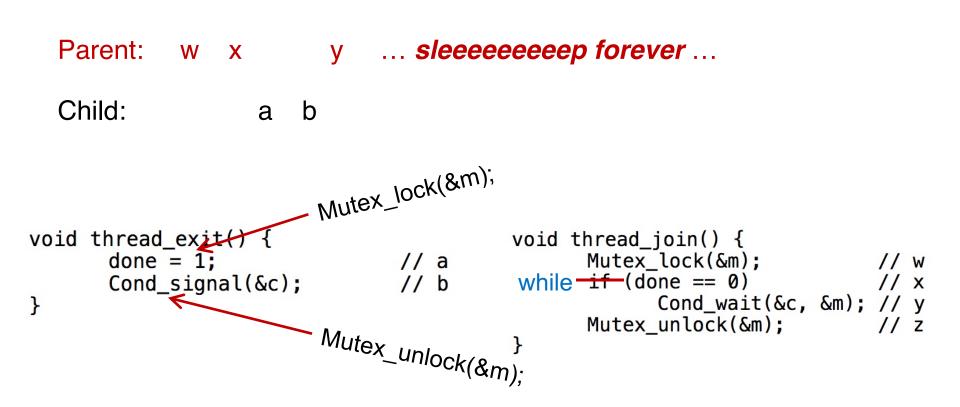
Parent:	x y	sleeeeeeeep forever
Child: a b c		
<pre>void thread_exit() { Mutex_lock(&m) Cond_signal(&c Mutex_unlock(& }</pre>);	<pre>void thread_join() { // a Mutex_lock(&m); // x // b Cond_wait(&c, &m); // y // c Mutex_unlock(&m); // z }</pre>

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
```

```
}
```

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

```
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?

<pre>roid *consumer(void *arg) { C1 ru int i;</pre>	C1 running			
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>				
}				

V

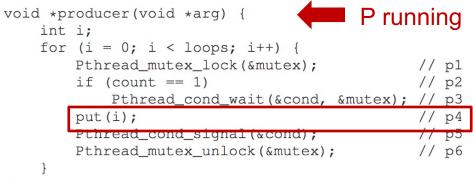
T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get

<pre>void *consumer(void *arg) {</pre>	
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>	
}	

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

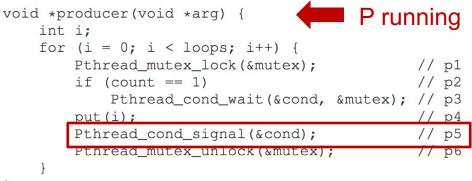
T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	0 0
	Sleep		Ready	p2	Running	0	

<pre>void *consumer(void *arg) {</pre>	
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
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

<pre>void *consumer(void *arg) {</pre>	
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
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	0 0
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken

<pre>void *consumer(void *arg) {</pre>	
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>	
}	

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 11 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
	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 ANNO 111.111.111.1

<pre>void *consumer(void *arg) {</pre>	
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>	
}	

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 11 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
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	and a state state that

<pre>void *consumer(void *arg) { int i;</pre>	C1 runnable	int i;
for (i = 0; i < loops; i++) {		for (i = 0; i < loops; i++) {
<pre>Pthread_mutex_lock(&mutex);</pre>	// c1	Pthread_mutex_lock(&mutex); // p
if (count $== 0$)	// c2	if (count == 1) // p.
Pthread_cond_wait(&cond,	&mutex); // c3	<pre>Pthread_cond_wait(&cond, &mutex); // p</pre>
<pre>int tmp = get();</pre>	// c4	put(i); // p
<pre>Pthread_cond_signal(&cond);</pre>	// c5	Pthread_cond_signal(&cond); // p.
<pre>Pthread_mutex_unlock(&mutex);</pre>	// c6	<pre>Pthread_mutex_unlock(&mutex); // p</pre>
<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	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

<pre>void *consumer(void *arg) {</pre>	nning	7
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	
int tmp = get();	// 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

<pre>void *consumer(void *arg) {</pre>	nning	voi	d *p: int	
for $(i = 0; i < loops; i++) $ {			for	(i
<pre>Pthread_mutex_lock(&mutex);</pre>	// c1			Pth
if (count $== 0$)	// c2			if
<pre>Pthread_cond_wait(&cond, &mutex);</pre>	// c3			
<pre>int tmp = get();</pre>	// c4		-	put
<pre>Pthread_cond_signal(&cond);</pre>	// c5			Pth
<pre>Pthread_mutex_unlock(&mutex);</pre>	// c6			Pth
<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

<pre>d *consumer(void *arg) { C2 ru int i; for (i = 0; i < loops; i++) {</pre>	nning	voi	.d i
Pthread_mutex_lock (&mutex);	// c1		-
if (count == 0)	// c2		
Pthread_cond_wait(&cond, &mutex);	// 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
	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

void *consumer(void *arg) { C2 rur	nning	1
int i;	Ŭ	
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
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	

<pre>void *consumer(void *arg) { int i;</pre>	C1 running	<pre>void *producer(void *arg) { int i;</pre>
for (i = 0; i < loops; i++) {		for (i = 0; i < loops; i++) {
<pre>Pthread_mutex_lock(&mutex);</pre>	// c1	Pthread_mutex_lock(&mutex); // p1
if (count $==$ 0)	// c2	if (count == 1) // p2
Pthread_cond_wait(&cond,	&mutex); // c3	Pthread_cond_wait(&cond, &mutex); // p3
<pre>int tmp = get();</pre>	// c4	put(i); // p4
Pthread_cond_signal(&cond);	// c5	Pthread_cond_signal(&cond); // p5
<pre>Pthread_mutex_unlock(&mutex); printf("%d\n", tmp);</pre>	// c6	Pthread_mutex_unlock(&mutex); // 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	
		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
                                                                           35
27
```

	l *c int		(void *arg) {		C1 runr	ning	vo	int i;	er(void *a		
	for	(i = 0)	; i < loops;	i++) {						ops; i++) {	
	<u>г</u>	Pthrea	d_mutex_lock	(&mutex);		// c1				lock(&mutex);	// p1
	_ I	while	(count == 0)			// c2		whil	e (count ==	= 1)	// p2
	_ I	Pt	hread_cond_wa	ait (&cond,	&mutex);	// c3		1	Pthread_com	nd_wait(&cond, &mutex);	// p3
		int tm	p = get();			// c4		put (i);		// p4
		Pthrea	d_cond_signal	(&cond);		// c5		Pthr	ead_cond_s:	ignal(&cond);	// p5
			d_mutex_unloc		;	// c6		Pthr	ead_mutex_u	unlock(&mutex);	// p6
			("%d\n", tmp)					}			
	}	I	, , , , , , , , , , , , , , , , , , , ,				}				
}		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	
		CO	Sleep		Reauy			Reauy	U	Notifing to get	

voi	int	: i;	(void *arg)		C2 runr	ning	void *produc int i;			
	for		; i < loops;				for $(i = 0; i < loops; i++) $ {			11 7
		Pthrea	d_mutex_lock	(&mutex)	;	// c1	Pthread_mutex_lock(&mutex);			// p1
		while	(count == 0)			// c2	while (count == 1)			// p2
		Pt	hread_cond_wa	ait (&cond	d, &mutex);	// c3	<pre>Pthread_cond_wait(&cond, &mutex);</pre>			
			p = get();			// c4	put(i);			
		Pthread_cond_signal(&cond); // c					Pthr	ead_cond_s:	ignal(&cond);	// p5
	Pthread_mutex_unlock(&mutex);					// c6	Pthr	ead_mutex_u	unlock(&mutex);	// p6
	<pre>printf("%d\n", tmp);</pre>					// 00	}			-
	l	PIINCI	(da(ii / cmp)	/			}			
1	ſ						,			
3		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	

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {</pre>
        Pthread_mutex_lock(&mutex);
                                               // c1
                                               // 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>	nir	ıg
<pre>Pthread_mutex_lock(&mutex); while (count == 1)</pre>	11	p1 p2
Pthread_cond_wait(&cond, &mutex);	//	р3
put(i);	//	p4
Pthread_cond_signal(&cond);	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++) {
       Pthread_mutex_lock(&mutex);
                                              // c1
                                              // 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);
    }
```

int		nin	g
101	<pre>(i = 0; i < loops; i++) { Pthread_mutex_lock(&mutex); while (count == 1) Pthread_cond_wait(&cond, &mutex); put(i);</pre>		p2
C	<pre>Pthread_cond_signal(&cond);</pre>	//	р5
_	Pthread_mutex_unlock(&mutex);	//	рб
}			

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	roducer(void *arg) { PS	eepi	ng
for	(i = 0; i < loops; i++) {		
	<pre>Pthread_mutex_lock(&mutex);</pre>	11	p1
_	while (count == 1)	11	p2
[Pthread cond wait(&cond, &mutex); //	ъЗ
	put(i);	11	p4
	Pthread_cond_signal(&cond);	11	p5
	<pre>Pthread_mutex_unlock(&mutex);</pre>	11	p6
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	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)

void	d *co int	C1 runr	ning	void						
	for	Pthrea	; i < loops; d_mutex_lock			// c1				
	– E		(count == 0)			// c2				
		Pt	hread_cond_wa	ait(&cond,	&mutex);	// c3				
			// c4							
	<pre>int tmp = get(); // c Pthread_cond_signal(&cond); // c</pre>									
	// c6									
			d_mutex_unloc ("%d\n", tmp)		<i>,</i>	// 00				
	}	princi		1			}			
}		T_{c1}	State	T_{c2}	State	$ T_p $				
	-	c 1	Running		Ready		I			
		c2	Running		Ready		J			
		c3	Sloop		Ready		T			

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	
	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

in	t i; pr (i = 0 Pthrea while Pt int tm Pthrea Pthrea	<pre>(void *arg) { i < loops; d_mutex_lock (count == 0) thread_cond_wa mp = get(); id_cond_signal d_mutex_unloc ["%d\n", tmp)</pre>	i++) { (&mutex) ait(&cond) ck(&mute:	d, &mutex);	ning // c1 // c2 // c3 // c4 // c5 // c6	Pthr whil put(Pthr	0; i < lo ead_mutex_ e (count = Pthread_co
}	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
		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
					1		

arg) { oops; i++) { _lock(&mutex); // p1 == 1) // p2 ond wait(&cond, &mutex); // p3 // p4 // p5 signal(&cond); // p6 _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	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

void *c int	onsumer(void *arg) { C1 run	ning	vo
	<pre>(i = 0; i < loops; i++) { Pthread_mutex_lock(&mutex); while (count == 0) Pthread_cond_wait(&cond, &mutex);</pre>	// c1 // c2 // c3	
	<pre>int tmp = get();</pre>	// c4	
L	Pthread_cond_signal(&cond);	// c5	
	<pre>Pthread_mutex_unlock(&mutex); printf("%d\n", tmp);</pre>	// c6	
}	T = State T = State		}

void *producer(void *arg) { int i; for (i = 0; i < loops; i++) {</pre> Pthread_mutex_lock(&mutex); // p1 while (count == 1) // p2
Pthread_cond_wait(&cond, &mutex); // p3 put(i); // p4 Pthread_cond_signal(&cond); // p5 Pthread_mutex_unlock(&mutex); // p6 }

T_{c1}	State	T _{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	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 *arg) {		C1 slee	<pre>void *producer(void *arg) { int i;</pre>				
	it i;	1			1 0	for $(i = 0; i < loops; i++)$ {			
IO		; i < loops; d_mutex_lock		;	Pthread_mutex_lock(&mutex);				
		(count == 0)	(amutex)	ř	while (count == 1)				
		hread_cond_wa	ait(&con		Pthread cond wait (&cond, &mutex);				
		p = get();			put(i);				
	Pthrea	d_cond_signal	(&cond)	;	// c5	<pre>Pthread_cond_signal(&cond);</pre>			
		d_mutex_unloc		x);	// c6	Pthread_mutex_unlock(&mutex);			
,	printf	("%d\n", tmp)	;			}			
}	T	CL	- m	CL		Clark	0	G	
,	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)	
	c2	Running		Sleep	P	Sleep	1	Recheck condition	
	c4	Running		Sleep		Sleep	0	T_{c1} grabs data	
	c5	0		-		-	0	0	
		Running		Ready		Sleep		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	

4

// p1

// p2

// p3 // p4 // p5

// p6

[}]

voi			(void *arg)	{	C2 runn	<pre>void *producer(void *arg) {</pre>				
		ti;			021011	int i;				
	fo		; i < loops;			for $(i = 0; i < loops; i++)$ {			// p1	
			d_mutex_lock	(&mutex)		<pre>Pthread_mutex_lock(&mutex); while (count == 1)</pre>			// p1	
	<pre>while (count == 0) // c2 Pthread_cond_wait(&cond, &mutex); // c3</pre>								nd_wait(&cond, &mutex)	
			p = get();	are (acon	a, anacex,,				// p4	
			d_cond_signal	l(&cond)		Pthread_cond_signal(&cond);			// p5	
		Pthrea	d_mutex_unlo	ck (&mute:	x);	Pthread_mutex_unlock(&mutex); /			// p6	
		printf	("%d\n", tmp);		}				
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	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	гu	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	Cops. Hoke 1 ₆₂	
		c1	Running				Sleep	0		
		c2	0		Ready			0		
			Running		Ready		Sleep		Nothing to get	
		c3	Sleep	-2	Ready		Sleep	0	Nothing to get	
			Sleep	c2	Running		Sleep	0	F 1	-5
			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
        Pthread_mutex_unlock(&mutex);
                                               // c6
        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	-6
	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
         }
```

27

ł

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
                                                                          48
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
20
    void rwlock_release_readlock(rwlock_t *rw) {
21
                                                                                 be 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
                                                                                               56
      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?

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 4 Philosopher 3 Philosopher 3 Philosopher 2

Shared data

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

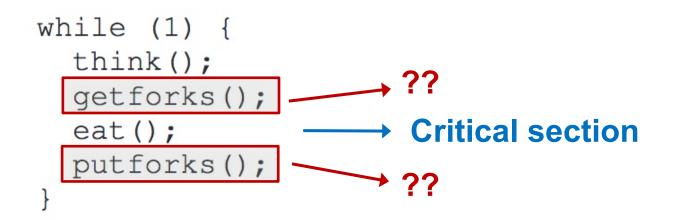
while(food available) {pick up 2 adj. forks; eat; put down forks; think awhile;

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 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])

sem wait(fork[0])

Thread 0 Interleaving

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 1

```
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[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])

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

Thread 0

Interleaving

Thread 1

sem wait(fork[1])

sem wait(fork[0])

sem signal(fork[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_signal(fork[0])

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

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

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

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

Dijkstra's Solution: Break the Circular Wait Condition

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