

# Recap

## Pthread Library

- `pthread_create`, `pthread_exit`, `pthread_join`
- `pthread_mutex_lock`, `pthread_mutex_unlock`

## Critical section: mutual exclusion

## Lock implementation

- disabling interrupt
- Peterson's algorithm

# Peterson's Algorithm – A Software Solution That Works!

```
int flag[2];
int turn;

void init() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}

void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while it's not your turn
}

void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

Case 1: sequential execution

For example, Proc0.lock();Proc1.lock().

Proc0 finds flag[1]==0, so it locks.

Proc1 can lock only after Proc0 unlocks.

Case 2: concurrent execution

time	P0	P1
t	Flag[0]=1	Flag[1]=1
t+1	turn=1	
t+2	(block)	turn=0
t+3	lock!	(block)
t+4	unlock	(block)
t+5		lock!

Can be extended to more than 2 processes!

# Disadvantage of Peterson's Algorithm

Peterson's solution does not work on modern computer architectures

To improve performance, processors can reorder instructions that have no dependencies

What happens if assignments to flag and turn are reordered?

# Hardware Support – Test-and-Set

Common hardware support is a test-and-set instruction

```
1  int TestAndSet(int *old_ptr, int new) {  
2      int old = *old_ptr; // fetch old value at old_ptr  
3      *old_ptr = new;      // store 'new' into old_ptr  
4      return old;          // return the old value  
5  }
```



Operation of test-and-set is shown in code above, but the important point is test-and-set is not software, it is an **atomic instruction** (cannot be interrupted) so therefore no race condition possible

# How to Use Test-and-Set to Build a Lock?

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

# Problem: Performance of Spinning

All the approaches we have seen are **spin locks**, a waiting thread keeps checking until the lock is available

Uses CPU for indefinite amount of time

- On single CPU machine, wait for time-slice to expire

- N threads contending for lock wait N-1 time slices

# Solution to Spinning - Yield

Solution is that waiting thread should voluntarily give up CPU

```
1  void init() {
2      flag = 0;
3  }
4
5  void lock() {
6      while (TestAndSet(&flag, 1) == 1)
7          yield(); // give up the CPU
8  }
9
10 void unlock() {
11     flag = 0;
12 }
```

# Problem: Fairness

So far, when multiple threads contending for lock the winner is up to chance  
Which ever one executes TestAndSet first

A more controlled mechanism is a FIFO queue for thread waiting on lock



# Solution to Fairness - Queue

park() is system call to put thread to sleep until unpark(tid) is called

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, getpid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock
33                                     // (for next thread!)
34     m->guard = 0;
35 }
```

# L16: Condition Variables

(based on Ch. 30)

# Condition Variables



We have seen **spinning loops** are very inefficient

It is a common need for a thread to wait for another thread to complete some task

For example, `pthread_join()`

How to provide an efficient wait?

# Problem

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

# Simple Solution – Spinning Loop

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

# Condition Variable

A **condition variable** is a queue that threads can put themselves in when **waiting** for some **condition**

Some other thread can wake one (or more) of the waiting threads by **signaling** the **condition**

# pthread Example

```
pthread_cond_signal(  
    pthread_cond_t *c)
```

Signal on the condition, waking waiting threads

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

Waits for signal to condition c, releases the mutex lock m while waiting

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

# Correct Usage

`pthread_cond_wait` gives up lock when thread goes to blocked state and requires it before return

```
19     Pthread_mutex_lock (&m) ;  
20     while (done == 0)  
21         Pthread_cond_wait (&c, &m) ;  
22     Pthread_mutex_unlock (&m) ;
```



# Why Put Signal and Wait in Critical Section?

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

Wrong  
Implementation

Consider:

T1 line 7 – it has done=0, so takes the true branch

Context switch to T2

T2 line 2

T2 line 3 – calls signal before the other thread is waiting

Context switch to T1

T1 line 8 – now T1 is stuck in wait

# Classic Concurrency Problems

Concurrency makes reasoning about code very difficult

There are several classic (example) problems that have been created to illustrate issues in concurrency and common solutions

It is a good idea to study the solutions to these problems and use them as patterns in your own code

# The Producer/Consumer (Bounded Buffer) Problem

One or more producer threads generate data items and place them in a buffer

One or more consumers grab items from the buffer and consume them

Common example: a pipe between processes acts as a buffer with concurrent producer and consumer

```
1  void *producer(void *arg) {
2      int i;
3      int loops = (int) arg;
4      for (i = 0; i < loops; i++) {
5          put(i);
6      }
7  }
8
9  void *consumer(void *arg) {
10     while (1) {
11         int tmp = get();
12         printf("%d\n", tmp);
13     }
14 }
```

A non-thread-safe version

# Attempt 1 (Broken)

Need some way to prevent  
calling put of full buffer and  
get on empty buffer

This attempt is broken

To understand why, need to  
know one more detail about  
condition variables...

```
1  int loops; // must initialize somewhere...
2  cond_t  cond;
3  mutex_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          Pthread_mutex_lock(&mutex);          // p1
9          if (count == 1)                      // p2
10             Pthread_cond_wait(&cond, &mutex); // p3
11             put(i);                          // p4
12             Pthread_cond_signal(&cond);      // p5
13             Pthread_mutex_unlock(&mutex);    // p6
14         }
15     }
16
17     void *consumer(void *arg) {
18         int i;
19         for (i = 0; i < loops; i++) {
20             Pthread_mutex_lock(&mutex);      // c1
21             if (count == 0)                  // c2
22                 Pthread_cond_wait(&cond, &mutex); // c3
23             int tmp = get();                 // c4
24             Pthread_cond_signal(&cond);      // c5
25             Pthread_mutex_unlock(&mutex);    // c6
26             printf("%d\n", tmp);
27         }
28     }
```

# Mesa Semantics for Condition Variables

The most common implementations of condition variables (including pthreads) follow **mesa semantics**

When a sleeping (waiting) thread is woken, it is placed in a ready queue

It does not require the mutex lock to be in the queue

Only when it is its turn to run does it acquire the lock

# Attempt 1 (Broken)

Example of where Mesa semantics  
can lead to issues

Consider 2 consumers and 1  
producer

- Consumer1 waits for condition
- Producer signals which puts consumer1 in ready queue
- **Before consumer1 runs consumer2 consumes buffer**
- Consumer1 is set to running and tries to consume from empty buffer (error)

```
1  int loops; // must initialize somewhere...
2  cond_t  cond;
3  mutex_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          Pthread_mutex_lock(&mutex);          // p1
9          if (count == 1)                      // p2
10             Pthread_cond_wait(&cond, &mutex); // p3
11             put(i);                          // p4
12             Pthread_cond_signal(&cond);      // p5
13             Pthread_mutex_unlock(&mutex);    // p6
14         }
15     }
16
17  void *consumer(void *arg) {
18      int i;
19      for (i = 0; i < loops; i++) {
20          Pthread_mutex_lock(&mutex);          // c1
21          if (count == 0)                      // c2
22             Pthread_cond_wait(&cond, &mutex); // c3
23             int tmp = get();                  // c4
24             Pthread_cond_signal(&cond);      // c5
25             Pthread_mutex_unlock(&mutex);    // c6
26             printf("%d\n", tmp);
27         }
28     }
```

## Attempt 2 (Better, Still Broken)

Consider 2 consumers and 1 producer

What can go wrong?

```
1  int loops;
2  cond_t  cond;
3  mutex_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          Pthread_mutex_lock(&mutex);           // p1
9          while (count == 1)                    // p2
10             Pthread_cond_wait(&cond, &mutex); // p3
11             put(i);                             // p4
12             Pthread_cond_signal(&cond);         // p5
13             Pthread_mutex_unlock(&mutex);       // p6
14         }
15     }
16
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex);           // c1
21         while (count == 0)                    // c2
22             Pthread_cond_wait(&cond, &mutex); // c3
23         int tmp = get();                       // c4
24         Pthread_cond_signal(&cond);           // c5
25         Pthread_mutex_unlock(&mutex);         // c6
26         printf("%d\n", tmp);
27     }
28 }
```

## Attempt 2 (Better, Still Broken)

Consider 2 consumers and 1 producer

Consumer1 and Consumer2 both go into wait

Producer adds to buffer and waits

Consumer1 consumes from buffer and **signals**

Consumer2 **gets the signal**, it tries to read from empty buffer (error)

```
1  int loops;
2  cond_t  cond;
3  mutex_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          Pthread_mutex_lock(&mutex);          // p1
9          while (count == 1)                    // p2
10             Pthread_cond_wait(&cond, &mutex); // p3
11          put(i);                               // p4
12          Pthread_cond_signal(&cond);          // p5
13          Pthread_mutex_unlock(&mutex);        // p6
14      }
15  }
16
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex);          // c1
21         while (count == 0)                    // c2
22             Pthread_cond_wait(&cond, &mutex); // c3
23         int tmp = get();                      // c4
24         Pthread_cond_signal(&cond);          // c5
25         Pthread_mutex_unlock(&mutex);        // c6
26         printf("%d\n", tmp);
27     }
28 }
```



# Working Version

Need condition variables to signal both empty and full

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