Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department

Zeinab Zali

Synchronization-Condition Variables-monitors

Condition Variable

- there are many cases where a thread wishes to check whether a condition is true before continuing its execution
- To wait for a condition to become true, a thread can make use of what is known as a condition variable
- A condition variable is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition)
- some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition)

CV definitions and routines

- pthread_cond_t c
- pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m): The wait() call is executed when a thread wishes to put itself to sleep
  - The responsibility of wait() is to release the lock and put the calling thread to sleep (atomically); when the thread wakes up (after some other thread has signaled it), it must re-acquire the lock before returning to the caller.
- pthread_cond_signal(pthread_cond_t *c): the signal() call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition

CV example 1

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

Desired Output:

```
parent: begin
child
parent: end
```

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

void thr_exit() {
}

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

void thr_join() {
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

What is the problem with this solution?

```
void thr_exit() {
        Pthread_mutex_lock(&m);
done =1;
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void thr_join() {
        Pthread_mutex_lock(&m);
while (done == 0)
        Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

What is the problem with this solution?

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

Producer/Consumer Problem

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
                                                  // p1
                                                  // p2
                                                  // p3
                                                  // p4
                                                  // p5
                                                  // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
                                                  // c1
                                                  // c2
                                                  // c3
                                                  // c4
                                                  // c5
                                                  // c6

    }
}
```

Producer/Consumer Problem

```
int buffer;
int count = 0; // initially, empty

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

Producer/Consumer Problem

```
int loops; // must initialize somewhere...
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);
    }
}
```

Producer/Consumer broken solution with if

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

```
int loops;
cond_t  cond;
mutex_t mutex;

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

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

```
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                  // p1
        while (count == 1)                           // p2
            Pthread_cond_wait(&cond, &mutex);        // p3
```

TIP: USE WHILE (NOT IF) FOR CONDITIONS

When checking for a condition in a multi-threaded program, using a `while` loop is always correct; using an `if` statement only might be, depending on the semantics of signaling. Thus, always use `while` and your code will behave as expected.

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

Producer/Consumer still a broken solution

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Producer/Consumer with buffer size MAX

```
cond_t empty, fill;
mutex_t mutex;

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

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

Producer/Consumer with buffer size MAX

```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

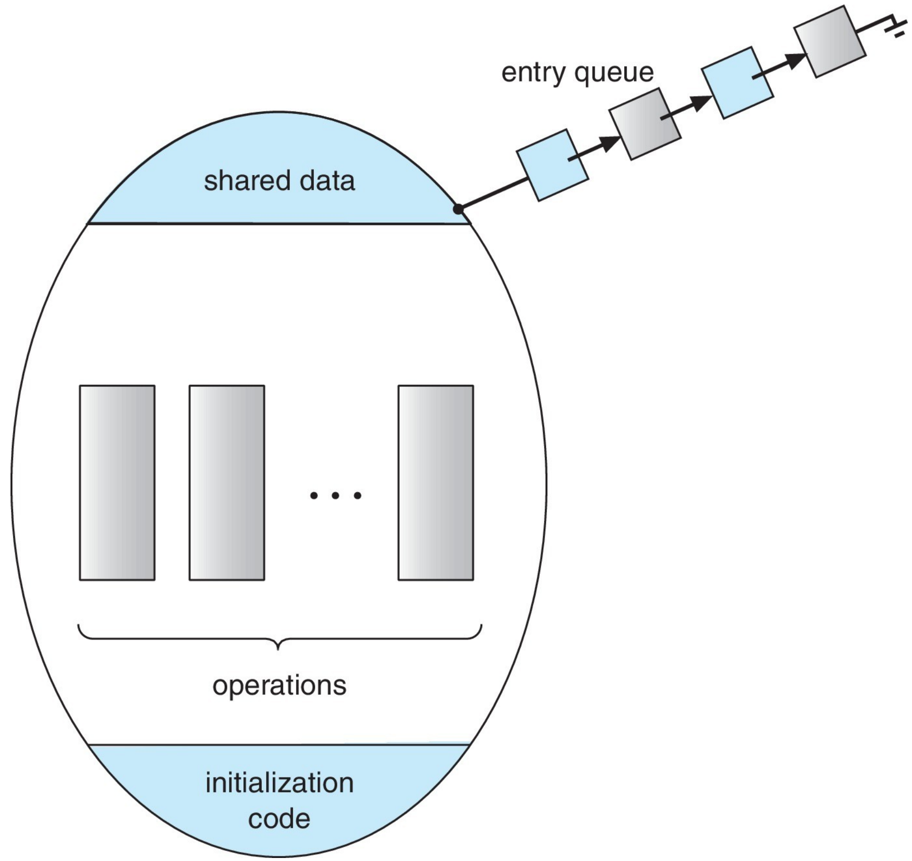
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure P2 (…) { …. }

  procedure Pn (…) {……}

   initialization code (…) { … }
}
```

Schematic view of a Monitor



Monitor Implementation Using Semaphores

- Variables

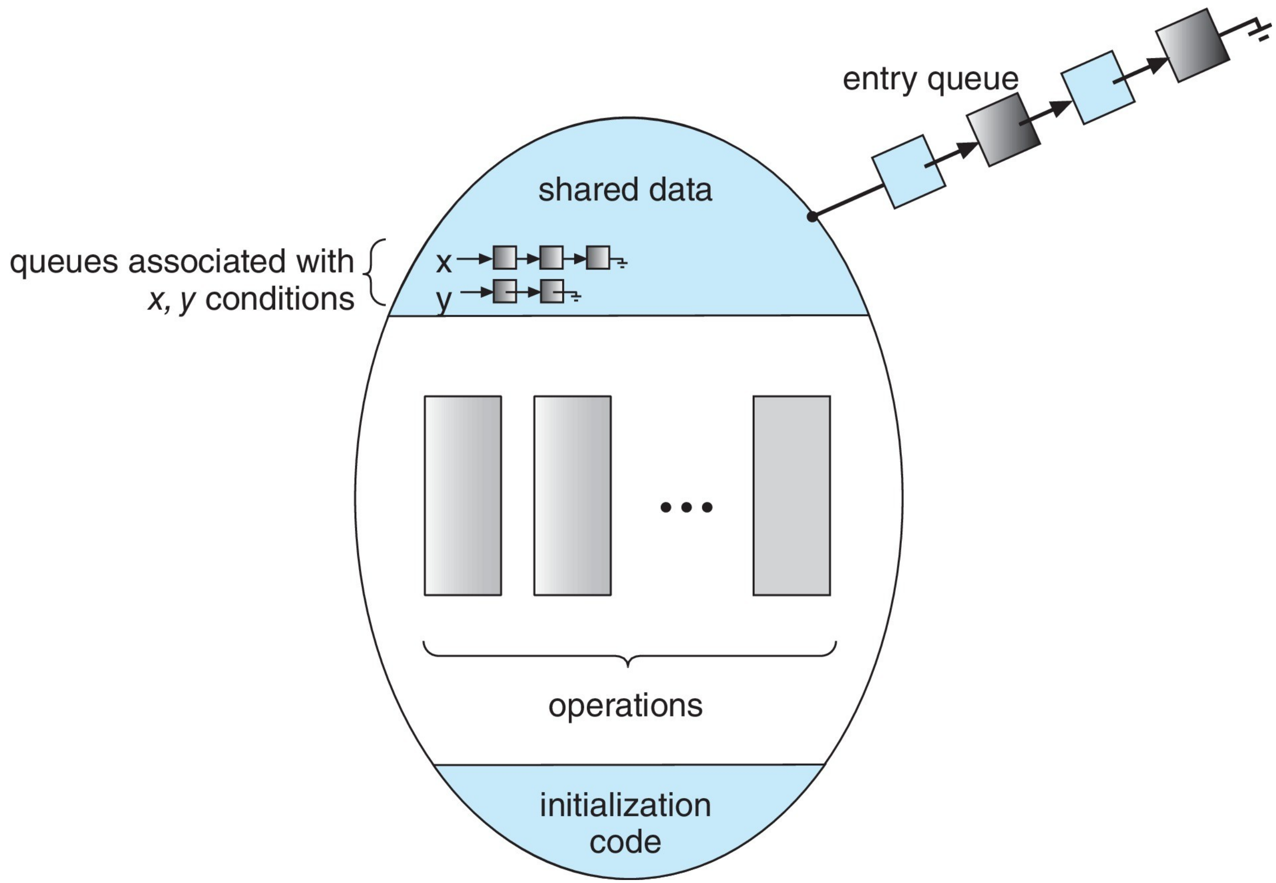
```
semaphore mutex
mutex = 1
```

- Each procedure ***P*** is replaced by

```
wait(mutex);
    …
        body of P;
    …
signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor with Condition Variables



Usage of Condition Variable Example

- Consider P_1 and P_2 that need to execute two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a monitor with two procedures F_1 and F_2 that are invoked by P_1 and P_2 respectively
  - One condition variable “x” initialized to 0
  - One Boolean variable “done”
  - **F1:**

```
    S1;
    done = true;
    x.signal();
```

  - **F2:**

```
    if done = false
       x.wait()
    S2;
```

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
         }

   void putdown (int i) {
   }
```

- Each philosopher "i" invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

     /** EAT **/

DiningPhilosophers.putdown(i);
```

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self[i].wait;
   }

   void putdown (int i) {
          state[i] = THINKING;
                   // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
   }
```

Solution to Dining Philosophers (Cont.)

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                 state[i] = EATING ;
            self[i].signal () ;
            }
    }

        initialization_code() {
           for (int i = 0; i < 5; i++)
               state[i] = THINKING;
        }
}
```

Solution to Dining Philosophers (Cont.)

- Each philosopher "i" invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

    /** EAT **/

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

End of Chapter 6