# Ch5: Synchronization

## Critical Section

- Parts of the program where shared resource is accessed, so they need to be protected
- Conditions:
  - Mutual Exclusion
  - Progress
  - Bounded Waiting

## Mutex Lock

### Peterson's Solution

```
/*
// Set flag[0], flag[1] to be false to allow progress
// turn solves the trouble that both process are checked not ready but
*/

// p0:
do {
    flag[0] = true; turn = 1;
    while(flag[1]&&turn==1); // this is a dummy loop
    //critical section
    flag[0] = false;
    //remaining section
} while(1);

// p1:
do {
    flag[1] = true; turn = 0;
    while(flag[0] && turn == 0); // dummy loop
    flag[1] = false;
    // remaining section
} while(1);
```

### Hardware Support for Synchronization

- atomic hardware instructions
  - Test memory word and set value
  - Swap contents of two memory words

## TestAndSet Instruction

```
// atomic means finish immediately
// if *lock = true -> TestAndSet() = true; *lock = true;
// if *lock = false -> TestAndSet() = false; *lock = true;

boolean TestAndSet(boolean *lock) {
    boolean rv = *lock;
    *lock = true;
    return rv;
}
```

```
// Initially set lock to be false, so that it can escape the dummy loop
// Example:
do {
    while(TestAndSet(lock));
    lock = false;
} while(1);
```

## Swap Instruction

```
void Swap(boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

// critical section

do {
    key = true;
    while(key == true)
        Swap(&lock, &key); // dummy loop until lock is false;
    // critical section
    lock = false;
    // remaining section
} while(1);
```

# Semaphore

- semaphore s: integer value
- two operations: **wait()** and **signal()**
- **Counting Semaphores**: value could be positive integers
    - multiple copies of the shared value (e.g. readLock)
- **Binary Semaphores**: could only be 0 or 1

```
//pseudo code
process:
    wait(s);
    critical section
    signal(s);
```

- must guarantee no two processes access S at the same time

### Busy Waiting

```
wait(S) {
    while(S <= 0);
    S--;
}
```

### No Busy Waiting

- Two operations: **block**, **wakeup**

```
semaphore {
    int value;
    Queue q;
}

wait(semaphore S) {
    S.value--;
    if (S.value < 0){   // it means that there's no sufficient resource
        // add this process/thread to the queue
        block() //blocks this process and hands over the control to OS
    }
}

signal(semaphore S) {
    S.value++;
    if (s.value <= 0) { // it means that there are waiting processes
        //process P = remove a process from the waiting queue
        wakeup(p);
    }
}
```

### DeadLock

- A situation where no process could proceed because each waits for another to release a lock

## Classical Synchronization Problems

- Producer-Consumer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

## Producer-Consumer Problem

- multiple producers and consumers share a single buffer of size N
- Semaphores:
  - mutex(initialized to 1): protect access to critical sections
  - full(initialized to 0): to count the occupied slots in the buffer
  - empty(initialized to N): to count the available slots in the buffer

```
// Producer:
do {
    // produce an item
    wait(empty);
    wait(mutex);
    // add the item to the buffer
    signal(mutex);
    signal(full);
} while(true);

// Consumer:
do {
    wait(full);
    wait(mutex);
    // remove an item from the buffer
    signal(mutex);
    signal(empty);
} while(true);
```

## Readers Writers Problem

- allow multiple reads at the same time, only one single writer access at the same time
- Solution:
  - **mutex** initialized to 1
  - **wrt** initialized to 1
  - shared integer variable ***readcount*** initialized to 0

```
// pseudo code

// writer:
do {
    wait(wrt);
    //writing is performed
    signal(wrt);
} while(true);

// reader:
do {
    wait(mutex);
    // begin of CS
    readcount++
```

```
    if (readcount == 1) wait(wrt); // only the first reader need to worry about
the writer
    // end of cs
    signal(mutex);
    // reading is performed here
    wait(mutex);
    readcount--;
    if (readcount == 0) signal(wrt); // only the last reader to leave need to
worry about the writer
    signal(mutex);
}while(true);
```

## Dining Philosopher Problem

```
// possible to cause deadlocks
do {
    wait(chopstick[i]);
    wait(chopstick[i+1]%5);
    // eat
    signal(chopstick[i]);
    signal(chopstick[i+1]%5);
    // think
} while(true);
```

# Conditional Variables

Conditional variable: **waiting** on the condition & **signaling** on the condition
Declaring the condition variable: `Pthread_cond_t c;`
Example:

```
// producer&consumer problem
// conditional variable, as its name, it is a condition for a thread to continue
executing, in the producer-consumer problem, then the condition is &fill and
&empty

void *producer(void* arg) {
    int i;
    for (i = 0; i < loops; i++) { // number of loops to go through
        Pthread_mutex_lock(&mutex); // the mutex lock for CS

        // begin of CS
        while (count == 1) // if the buffer is not empty, then wait
            Pthread_cond_wait(&empty, &mutex); // wait for both cond_v
        put(i); // produce
        Pthread_cond_signal(&fill); // signal the cond_v
        // end of CS

        Pthread_mutex_unlock(&mutex); // give back the lock
    }
}

void* consumer(void* arg) {
```

```
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);

        // begin of CS
        while (count == 0) // if the buffer is empty, then wait
            Pthread_cond_wait(&fill,&mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        // end of CS

        Pthread_mutex_unlock(&mutex);
    }
}

// this implementation allows consumers to wake up producers, and vice versa
(they wait in separate queues i.e. &fill queue and &empty queue; consumers should
not wake up consumers, same for producers)
// improvement for more concurrency and efficiency => Add more buffer slots:
// 1. to allow concurrent production on consuming and producing to take place
// 2.reduces context switches
```