CSE 156/L
Pthreads Example

Producer-Consumer Example

- Main thread spawns producer and consumer threads, then waits for them to complete
- Producer and consumer communicate through shared FIFO buffer
- Problems
  - Data races: requires mutual exclusion
  - Deadlocks
  - Busy waits

Pthreads Example: Initialization

```
#include <pthread.h>
#include <stdio.h>

#define BUF_SIZE 10          //size of shared FIFO buffer
int buffer[BUF_SIZE];        //shared buffer
int write_ptr=0;             //place to add next element
int read_ptr=0;              //place to remove next elem
int num=0;                   //buffer occupancy

//thread prototypes
void *producer(void *param);
void *consumer(void *param);
```

main

```
main (int argc, char *argv[]) {
        pthread_t tid1, tid2;          // thread identifiers
        int i;
        //create the producer and consumer threads
        if (pthread_create(&tid1, NULL, producer, NULL) != 0) {
                fprintf (stderr, "Unable to create producer    thread\n");
                exit (1);
        }
        if (pthread_create(&tid2, NULL, consumer, NULL) != 0) {
                fprintf (stderr, "Unable to create consumer thread\n");
                exit (1);
        }
        //wait for created thread to exit
        pthread_join(tid1,NULL);
        pthread_join(tid2,NULL);
        printf ("Parent quitting\n");
}
```

Pthreads Example: Producer

```
void *producer(void *param) {
        int i;
        for (i=1; i<=1000; i++) {
                //Insert into buffer
                if (num > BUF_SIZE) exit(1);    //error: overflow
                while (num == BUF_SIZE) {}     //block if buffer is full
                buffer[write_ptr] = i;
                write_ptr = (write_ptr+1) % BUF_SIZE;
                num++;
                printf("producer: inserted %d\n", i);  fflush(stdout);
        }
        printf("producer done\n");  fflush(stdout);
        return NULL;  // pthread_exit(0);          //this is optional
}
```

Pthreads Example: Consumer

```
void *consumer(void *param) {
        int i, count;
        for (count=0; count < 1000; count++) {
                if (num < 0) exit(1);       //error: underflow
                while (num == 0) {}        //block if buffer empty
                //if executing here, buffer not empty so remove element
                i = buffer[read_ptr];
                read_ptr = (read_ptr+1) % BUF_SIZE;
                num--;
                printf("Consumed value %d\n", i);  fflush(stdout);
        }
        printf("consumer done\n");  fflush(stdout);
        return NULL; // pthread_exit(0);  //this is optional
}
```

Adding mutex to fix data race

```
//mutex lock for buffer
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

Producer with Mutex

```
void *producer(void *param) {
        for (int i=1; i<=1000; i++) {                   // insert into buffer
                pthread_mutex_lock(&m);
                if (num > BUF_SIZE) exit(1);            //error: overflow
                while (num == BUF_SIZE) {}              //block if buffer is full
                buffer[write_ptr] = i;
                write_ptr = (write_ptr+1) % BUF_SIZE;
                num++;
                pthread_mutex_unlock(&m);
                printf("producer: inserted %d\n", i);  fflush(stdout);
        }
        printf("producer done\n");  fflush(stdout);
        return NULL;
}
```

Consumer with Mutex

```
void *consumer(void *param) {
        int i, count=0;
        for (count=0; count < 1000; count++) {
                pthread_mutex_lock (&m);
                if (num < 0) exit(1);       //error: underflow
                while (num == 0) {}         //block if buffer empty
                // if executing here, buffer not empty so remove element
                i = buffer[read_ptr];
                read_ptr = (read_ptr +1) % BUF_SIZE;
                num--;
                pthread_mutex_unlock (&m);
                printf("Consumed value %d\n", i); fflush(stdout);
        }
        printf("consumer done\n");  fflush(stdout);
        return NULL;
}  /* causes deadlock!! */
```

Avoiding Busy Wait

/* consumer waits on this cond var */

pthread_cond_t c_cons=PTHREAD_COND_INITIALIZER;

/* producer waits on this cond var */

pthread_cond_t c_prod=PTHREAD_COND_INITIALIZER;

Producer: Final Version

```
void *producer(void *param) {
        int i;
        for (i=1; i<=1000; i++) {    // Insert into buffer
                pthread_mutex_lock(&m);
                while (num == BUF_SIZE) //block if buffer is full
                        pthread_cond_wait(&c_prod, &m);
                if (num > BUF_SIZE) exit(1);    // error: overflow
                //if executing here, buffer not full so add element
                buffer[write_ptr] = i;
                write_ptr = (write_ptr +1) % BUF_SIZE;
                num++;
                pthread_mutex_unlock(&m);
                pthread_cond_signal(&c_cons);
                printf(”Producer inserted %d\n", i);  fflush(stdout);
        }
        printf("producer done\n"); fflush(stdout);
        return NULL;
}
```

Consumer: Final Version

```
void *consumer(void *param) {
        int i, count;
        for (count=0; count < 1000; count++) {
                pthread_mutex_lock (&m);
                while (num == 0)            // block if buffer empty */
                        pthread_cond_wait (&c_cons,&m);
                if (num < 0) exit(1);                //error: underflow
                //if executing here, buffer not empty so remove element
                i = buffer[read_ptr];
                read_ptr = (read_ptr +1) % BUF_SIZE;
                num--;
                pthread_mutex_unlock(&m);
                pthread_cond_signal(&c_prod);
                printf("Consumed value %d\n", i);  fflush(stdout);
        }
        printf("consumer done\n"); fflush(stdout);
        return NULL;
}
```

***

Semaphore Example: Initialization

```
#include <semaphore.h>

// … constants #define and thread prototypes producer(), consumer()
sem_t full;         // # of full
sem_t empty;     // # of empty
sem_t mutex;     // critical section

int main(int argc, char *argv[]) {
        // … setup init
        sem_init(&full, 0, 0);
        sem_init(&empty, 0, BUF_SIZE);
        sem_init(&mutex, 0, 1);

        // … thread creation and running
 }
```

Semaphore Example

```
#include <semaphore.h>
// …
```

```
void* consumer(void* p) {
   sem_wait(&full);
   sem_wait(&mutex);
   // … read entry

   sem_post(&mutex);
   sem_post(&empty);
}
```

```
void* producer(void* p) {
   sem_wait(&empty);
   sem_wait(&mutex);
   // ... write entry

   sem_post(&mutex);
   sem_post(&full);
}
```

***

Read/Write Lock (1)

- Allows multiple threads to read simultaneously, but prevents modifying data that's being read or written
- Many use cases where data is frequently read and not changing often
  - E.g., cache of recently accessed data, read by many threads
- Read cannot continue if write is in progress
- Write cannot continue if there are any reads or writes

Read/Write Lock (2)

- Varied behavior based on read precedence or write precedence to suit the application need
  - Potential for starvation?
  - E.g., writers may wait a long time if contention is high
  - Read precedence provides higher concurrency
- Can be implemented with a mutex and two conditions variables

Semaphore vs Mutex

- Mutex has a sense of ownership
  - If owned, owner is known (in most implementations)
  - Same thread lock/unlocks the mutex
- Semaphore has no knowledge of which thread is the owner, on which it is waiting (sleeping)
  - E.g., can't transfer scheduling priority
  - Hard for debug tools to track deadlocks or recursive locks