Lab 08 – Process Synchronization

- Data races

Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place. This is called a ***race condition***.

- Valgrind
- Mutex locks
- Semaphore
- Solution to Producer-Consumer Problem

Data Race (race.c)

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

int counter = 0;

void *func(void *params)
 {
   counter++;
   printf("%d\n", counter);
}

void main()
{
    pthread_t t1, t2;

    pthread_create(&t1, 0, func, 0);
    pthread_create(&t2, 0, func, 0);

    pthread_join(t1, 0);
    pthread_join(t2, 0);
}
```

Valgrind

- Download **valgrind_manual.pdf** from http://www.valgrind.org/
- Refer Page #134 (7.4 Detected errors: Data races)
- Can be used to detect data races with ***helgrind*** tool
- Compile race.c as shown below

```
gcc –g race.c –lpthread
valgrind --tool=helgrind ./a.out
```

```
==4156== Helgrind, a thread error detector
==4156== Copyright (C) 2007-2015, and GNU GPL'd, by OpenWorks LLP et al.
==4156== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4156== Command: ./a.out
==4156==
==4156== ---Thread-Announcement------------------------------------------
==4156==
==4156== Thread #3 was created
==4156==    at 0x51643DE: clone (clone.S:74)
==4156==    by 0x4E46149: create_thread (createthread.c:102)
==4156==    by 0x4E47E83: pthread_create@@GLIBC_2.2.5 (pthread_create.c:679)
==4156==    by 0x4C34BB7: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==4156==    by 0x400777: main (race.c:17)
==4156==
==4156== ---Thread-Announcement------------------------------------------
==4156==
==4156== Thread #2 was created
==4156==    at 0x51643DE: clone (clone.S:74)
==4156==    by 0x4E46149: create_thread (createthread.c:102)
==4156==    by 0x4E47E83: pthread_create@@GLIBC_2.2.5 (pthread_create.c:679)
==4156==    by 0x4C34BB7: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==4156==    by 0x40075C: main (race.c:16)
==4156==
==4156== ----------------------------------------------------------------
```

```
==4156==
==4156== Possible data race during read of size 4 at 0x601054 by thread #3
==4156== Locks held: none
==4156==    at 0x400702: func (race.c:8)
==4156==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==4156==    by 0x4E476B9: start_thread (pthread_create.c:333)
==4156==
==4156== This conflicts with a previous write of size 4 by thread #2
==4156== Locks held: none
==4156==    at 0x40070B: func (race.c:8)
==4156==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==4156==    by 0x4E476B9: start_thread (pthread_create.c:333)
==4156==  Address 0x601054 is 0 bytes inside data symbol "counter“
==4156==
==4156== ----------------------------------------------------------------
==4156==
==4156== Possible data race during write of size 4 at 0x601054 by thread #3
==4156== Locks held: none
==4156==    at 0x40070B: func (race.c:8)
==4156==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==4156==    by 0x4E476B9: start_thread (pthread_create.c:333)
==4156==
==4156== This conflicts with a previous write of size 4 by thread #2
==4156== Locks held: none
==4156==    at 0x40070B: func (race.c:8)
==4156==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==4156==    by 0x4E476B9: start_thread (pthread_create.c:333)
==4156==  Address 0x601054 is 0 bytes inside data symbol "counter“
==4156==
==4156==
==4156== For counts of detected and suppressed errors, rerun with: -v
==4156== Use --history-level=approx or =none to gain increased speed, at
==4156== the cost of reduced accuracy of conflicting-access information
==4156== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 12 from 12)
```

Mutex Locks

- Mutex lock can be acquired by 1 thread at a time
- Initialization of Mutex

```
pthread_mutex_init()
PTHREAD_MUTEX_INITIALIZER – statically defined mutex
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2;
pthread_mutex_init(&m2, 0);
```

m1 is same as m2 (look up the default values set for m1)

Mutex no longer required

```
pthread_mutex_destroy(&m1);
pthread_mutex_destroy(&m2);
```

- Mutex lock and unlock
- If a thread t1 calls lock when another thread t2 already holds the lock, the calling thread t1 will get blocked (or has to wait) until t2 releases the lock.

```
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t mutex;
volatile int counter = 0;
void *count(void *param)
{
  int *tid = (int *) param;
  for (int i = 0; i < 100; i++)
  {
    pthread_mutex_lock(&mutex);
    counter++;
    printf("Thread id - %d, Count = %i\n",
*tid, counter);
    pthread_mutex_unlock(&mutex);
  }
}
```

```
int main()
{
  pthread_t t1, t2;
  pthread_mutex_init(&mutex, 0);
  pthread_create(&t1, 0, count, (void *)&t1);
  pthread_create(&t2, 0, count, (void *)&t2);
  pthread_join(t1, 0);
  pthread_join(t2, 0);
  pthread_mutex_destroy(&mutex);

  return 0;
}
```

Semaphore

- Semaphore – Signaling and counting mechanism
- Allow threads to access a specified number of items
- In case of single item – semaphore is same as mutex (*recall in theory class we had discussed Boolean and Counting semaphore*)
- Initialization

```
sem_t semaphore;
sem_init (&semaphore, 0, 10);
```

2nd param – integer, indicates of the semaphore is shared between multiple processes or private to a process (0 means private)

- Header file is `semaphore.h`
- Destroy the semaphore – `sem_destroy(&semaphore);`
- Semaphore methods

`sem_wait()` or wait() as discussed in theory – tries to decrement the value until 0 beyond which calling thread waits until it becomes non-zero and then return

`sem_post()` or signal() as discussed in theory – tried to increment the semaphore

`sem_getvalue()` to get the current value of the semaphore into an integer (we can print the value to see)

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
sem_t semaphore;

void *func1(void *param)
{
  printf("Thread 1\n");
  sem_post(&semaphore); //signal
}

void *func2(void *param)
{
  sem_wait(&semaphore); //wait
  printf("Thread 2\n");
}
```

```
int main()
{
  pthread_t t[2];

  sem_init(&semaphore, 0, 1);

  pthread_create(&t[0], 0, func1, 0);
  pthread_create(&t[1], 0, func2, 0);

  pthread_join(t[0], 0);
  pthread_join(t[1], 0);

  sem_destroy(&semaphore);
}
```

Solution to Producer-Consumer problem

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NITERS (100)

int buf[5], f=0, r=0;

sem_t mutex, full, empty;
```

```c
void *produce(void *arg)
{
  int val1, val2;
  int i;

  printf("Producer created\n");

  for (int i = 0; i < NITERS; i++)
  {
    sem_wait( &empty );
    sem_wait( &mutex );
    buf[r] = i;
    r = (r + 1) % 5;
    sem_getvalue(&full, &val1);
    sem_getvalue(&empty, &val2);
    printf("P:: item = %d, full = %d, empty = %d\n", i, val1, val2);
    sem_post(&mutex);
    sem_post(&full);
  }
}
```

```
void *consume(void *arg)
{
  int item, i;
  printf("Consumer created\n");
  for (i = 0; i < NITERS; i++)
  {
    int val1, val2;

    sem_wait(&full);
    sem_wait( &mutex );

    item = buf[f];
    f = (f + 1) % 5;

    sem_getvalue(&full, &val1);
    sem_getvalue(&empty, &val2);
    printf("C:: item = %d, full = %d, empty = %d\n", item,
val1, val2);

    sem_post(&mutex);
    sem_post(&empty);

  }

}
```

```
int main()
{
  pthread_t t1, t2;

  sem_init(&mutex, 0, 1);

  sem_init(&full, 0, 0);
  sem_init(&empty, 0, 5);

  pthread_create(&t2, NULL,
consume, NULL);
  pthread_create(&t1, NULL, produce,
NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```