Threading & Concurrency

MP-3 Tutorial

Jhalak Banzal & Roja Sahoo

Threads

- Thread is a separate execution path.
- It is often referred to as a lightweight process.
- Each thread belongs to exactly one process.
- They share the same memory and resources as the program that created them. (main difference between processes)
- The process can be split down into so many threads - For example, in a browser, many tabs can be viewed as threads.
- MS Word uses many threads - formatting text from one thread, processing input from another thread.

Concurrency vs Parallelism

- Concurrency is about managing multiple instruction sequences at the same time, while parallelism is running multiple instruction sequences at the same time.
- Concurrency is achieved by using threading, while parallelism is achieved by using multitasking.
- Concurrency needs only one CPU Core, while parallelism needs more than one.

Process Synchronization

- Coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner.
- Necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems(Race condition).

Mutex

Mutex is a specific kind of binary semaphore that is used to provide a locking mechanism. It stands for

`Mutual Exclusion Object`

Each time a process requests a resource from the system, the system creates a mutex object with a unique name or ID.

Allows all processes to use the same resource, but the resource is accessed by one process at a time

- **Declaration**
  - `pthread_mutex_t lock;`
- **Initialization**
  - `pthread_mutex_init(&lock, NULL)`
- **Locking the mutex**
  - `pthread_mutex_lock(&lock)`
- **Unlocking the mutex**
  - `pthread_mutex_unlock(&lock);`
- **Destroying the mutex**
  - `pthread_mutex_destroy(&lock);`

Semaphore

It is used to control access to a resource that has multiple instances.

Signalling Mechanism

- `wait()` - decrements
- `signal()` - increments

Allow you to control multiple processes.

Semaphores allow multiple process threads to access a finite instance of a resource until a finite instance of the resource becomes available.

```
sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}
```

Condition Variables

- Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs.
- **`wait`**(condition, lock): release lock, put thread to sleep until condition is signaled; when thread wakes up again, re-acquire lock before returning.
- **`signal`**(condition, lock): if any threads are waiting on condition, wake up one of them. Caller must hold lock, which must be the same as the lock used in the wait call.
- **`broadcast`**(condition, lock): same as signal, except wake up all waiting threads.

```
char buffer[SIZE];
int count = 0, putIndex = 0, getIndex = 0;
struct lock l;
struct condition dataAvailable;
struct condition spaceAvailable;

lock_init(&l);
cond_init(&dataAvailable);
cond_init(&spaceAvailable);

void put(char c) {
    lock_acquire(&l);
    while (count == SIZE) {
        cond_wait(&spaceAvailable, &l);
    }
    count++;
    buffer[putIndex] = c;
    putIndex++;
    if (putIndex == SIZE) {
        putIndex = 0;
    }
    cond_signal(&dataAvailable, &l);
    lock_release(&l);
}

char get() {
    char c;
    lock_acquire(&l);
    while (count == 0) {
        cond_wait(&dataAvailable, &l);
    }
    count--;
    c = buffer[getIndex];
    getIndex++;
    if (getIndex == SIZE) {
        getIndex = 0;
    }
    cond_signal(&spaceAvailable, &l);
    lock_release(&l);
    return c;
}
```

Critical Section Problem - Dekkers

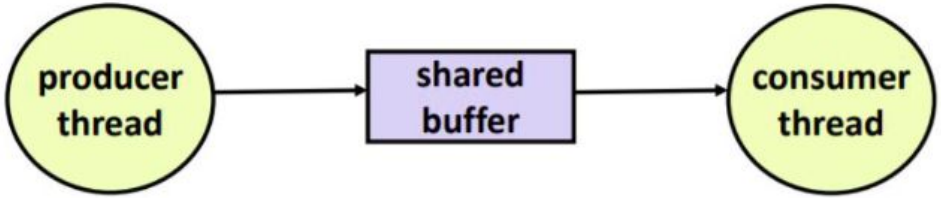
Partially correct solution to the Critical Section Problem.

The critical section problem refers to the problem of how to ensure that at most one process is executing its critical section at a given time.

V1 - If two processes try to access a resource simultaneously, Dekker's algorithm will allow only one of them to use it. It prevents conflict by imposing mutual exclusion, which means that only one process can access the resource and wait if another is doing so.

```
Thread1(){
  do {
    /* entry section
       wait until thread_no is 1 */
    while (thread_no == 2)
            continue;
    /* critical section
       exit section
       give access to the other thread */
    thread_no = 2;
    // remainder section
  } while (completed == false)
}
Thread2(){
  do {
    /* entry section
       wait until thread_no is 2 */
    while (thread_no == 1)
         continue;
    /* critical section
       exit section
       give access to the other thread */
    thread_no = 1;
    // remainder section
  } while (completed == false)
}
main(){
  int thread_no = 1;
  startThreads();
}
```

Producer Consumer - Petersons



- Producer waits for empty slot, inserts item in buffer, and notifies consumer
- Consumer waits for item, removes it from buffer, and notifies producer

Using Semaphores to solve

- Need two semaphores for signaling
  - One to track empty slots, and make producer wait if no more empty slots
  - One to track filled slots, and make consumer wait if no more filled slots
- One semaphore to act as mutex for buffer

Incorrect, why?

Initial Values

- `empty` - Capacity of buffer
- `Full` - 0

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);       // Line P0 (NEW LINE)
        sem_wait(&empty);       // Line P1
        put(i);                 // Line P2
        sem_post(&full);        // Line P3
        sem_post(&mutex);       // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);       // Line C0 (NEW LINE)
        sem_wait(&full);        // Line C1
        int tmp = get();        // Line C2
        sem_post(&empty);       // Line C3
        sem_post(&mutex);       // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

Correct Solution

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);       // Line P1
        sem_wait(&mutex);       // Line P1.5 (MUTEX HERE)
        put(i);                 // Line P2
        sem_post(&mutex);       // Line P2.5 (AND HERE)
        sem_post(&full);        // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);        // Line C1
        sem_wait(&mutex);       // Line C1.5 (MUTEX HERE)
        int tmp = get();        // Line C2
        sem_post(&mutex);       // Line C2.5 (AND HERE)
        sem_post(&empty);       // Line C3
        printf("%d\n", tmp);
    }
}
```

Dining Philosophers

Five philosophers sit around a circular table and alternate between thinking and eating. A bowl of noodles and five forks for each philosopher are placed at the center of the table.

There are certain conditions a philosopher must follow:

1. A philosopher must use both their right and left forks to eat.

2. A philosopher can only eat if both of his or her immediate left and right forks are available.

3. If the philosopher's immediate left and right forks are not available,

the philosopher places their (either left or right) forks on the table and resumes thinking.

Dining Philosophers

```
void Philosopher
{
 while(1)
  {
  take_fork[i];
  take_fork[ (i+1) % 5] ;

  EATING THE NOODLE

  put_fork[i];
  put_fork[ (i+1) % 5] ;

   THINKING
  }
}
```

```
void Philosopher
{
 while(true)
  {
  wait( F[i] );
  wait( F[ (i+1) % 5]);

  // EATING THE NOODLE

  signal( F[i] );
  signal( F[ (i+1) % 5] ) ;

   // THINKING
  }
}
```