CSE 4500: Operating Systems

Lecture 3

Process/Thread Synchronization

Background

- Parallelism can provide a distinct way of conceptualizing problems.
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers.
  - We can do so by having an integer *count* that keeps track of the number of full buffers.
  - Initially, *count* is set to 0.
  - It is incremented by the producer after it produces a new buffer
  - It is decremented by the consumer after it consumes a buffer.

Producer and Consumer

```
while (true) {
       /*  produce an item and put in nextProduced  */
         while (count == BUFFER_SIZE)
                  ; // do nothing
         buffer [in] = nextProduced;
         in = (in + 1) % BUFFER_SIZE;
         count++;
}
```

```
while (true)  {
            while (count == 0)
                ; // do nothing
                nextConsumed =  buffer[out];
                 out = (out + 1) % BUFFER_SIZE;
                     count--;
                   /*  consume the item in nextConsumed
      }
```

Race Condition

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2
- Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = count	{register1 = 5}
S1: producer execute register1 = register1 + 1	{register1 = 6}
S2: consumer execute register2 = count	{register2 = 5}
S3: consumer execute register2 = register2 - 1	{register2 = 4}
S4: producer execute count = register1	{count = 6 }
S5: consumer execute count = register2	{count = 4}

- Race conditions can occur when operations on shared variables are not ***atomic***.

Definitions

- **Synchronization**: using atomic operations to ensure cooperation between threads
- **Critical Section**: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.
  - Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - Progress: selecting a thread to enter cannot postpone indefinitely
  - Bounded waiting: before entering into the critical section
- **Lock**: prevents someone from doing something
  - Lock before accessing shared data
  - Unlock after accessing shared data
  - Wait if locked

Important idea: all synchronization involves waiting

Where to synchronize?

Programs	Shared Programs
API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

- We are going to implement various higher-level synchronization primitives
  - Need to provide primitives useful at user-level

Peterson's Solution

- A solution for two processes
- Assume that the LOAD and STORE instructions are atomic.
  - atomic == cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section.
  - flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
while (true) {
            flag[i] = TRUE;
            turn = j;
            while ( flag[j] && turn == j) //busy waiting
                        ;
            CRITICAL SECTION
              flag[i] = FALSE;
            REMAINDER SECTION
}
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
{
        boolean rv = *target;
        *target = TRUE;
        return rv:
}
```

- Solution: Shared Boolean variable *lock*, initialized to false.

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* do nothing
                //    critical section
        lock = FALSE;
                //      remainder section
}
```

Swap Instruction

- Definition:

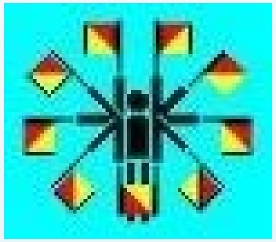
```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

- Solutions: Shared Boolean variable *lock* initialized to FALSE; Each process has a local Boolean variable *key*.

```
while (true)  {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );
        // critical section
        lock = FALSE;
        // remainder section
}
```

Semaphore

- Synchronization tool that does not require busy waiting
  - integer variable
  - Two standard operations
    - S.wait() => P()
    - S.signal() => V()
  - Less complicated
- Can only be accessed via two indivisible (atomic) operations



```
wait (S) {
  while S <= 0; // no-op
  S--;
}
```

```
signal (S) {
    S++;
}
```

Semaphore as General Synchronization Tool

- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Counting semaphore – integer value can range over an unrestricted domain
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

      Critical Section

    signal (S);

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

No Busy waiting Implementation

- With each semaphore there is an associated waiting queue.
  - value (of type integer)
  - a list of processes list (a list of PCBs)

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

No Busy waiting Implementation (Cont.)

- Implementation of wait:

```
wait (S){
        value--;
        if (value < 0) {
                add this process to waiting queue
                block();
        }
}
```

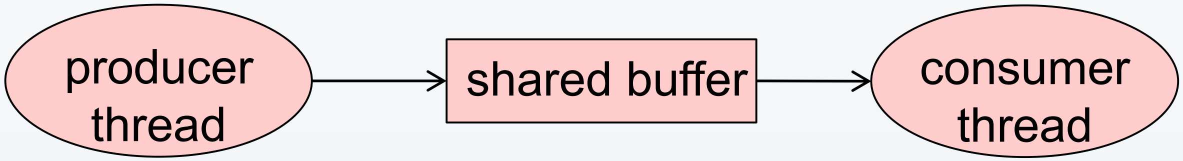
- Implementation of signal:

```
Signal (S){
        value++;
        if (value <= 0) {
                remove a process P from the waiting queue
                wakeup(P);
        }
}
```

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem

Bounded-Buffer Problem



- Common synchronization pattern:
  - Producer waits for slot, inserts item in buffer, and signals consumer
  - Consumer waits for item, removes it from buffer, and signals producer
- Examples
  - **Multimedia processing:**
    - Producer creates MPEG video frames, consumer renders the frames
  - **Graphical user interfaces**
    - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer.
    - Consumer retrieves events from buffer and paints the display.

Solving a Bounded-Buffer Problem

```
/* buffer.c - producer-consumer
on 1-element buffer */
#include <pthread.h>
#define N 5
void *producer(void *arg);
void *consumer(void *arg);
struct {
    int buf; /* shared var */
    sem_t full;     /* sems */
    sem_t empty;  /* sems */
} shared;
```

```
int main() {
    pthread_t tid_producer;
    pthread_t tid_consumer;
    /* initialize the semaphores */
    Sem_init(&shared.empty, 0, 1);
    Sem_init(&shared.full, 0, 0);
    /* create threads and wait */
    pthread_create(&tid_producer, NULL,
                            producer, NULL);
    pthread_create(&tid_consumer, NULL,
                            consumer, NULL);
    pthread_join(tid_producer, NULL);
    pthread_join(tid_consumer, NULL);
    exit(0);
};
```

Solving a Bounded-Buffer Problem

- **Initially empty=1, full=0.**

```
/* producer thread */
void *producer(void *arg) {
    int i, item;
    for (i=0; i<N; i++) { /* produce item */
        item = i;
        printf("produced %d\n", item);
        /* write item to buf */
       P(&shared.empty);
       shared.buf = item;
       V(&shared.full);
   }
   return NULL;
}
```

```
/* consumer thread */
void *consumer(void *arg) {
    int i, item;
    for (i=0; i<N; i++) { /* read item from buf */
       P(&shared.full);
       item = shared.buf;
       V(&shared.empty);
       /* consume item */
       printf("consumed %d\n", item);
   }
   return NULL;
}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set
  - Writers – can both read and write.
- Problems – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
  - First readers-writers problem: no reader should wait for other readers to finish simply because a writer is waiting.
  - Second readers-writers problem: once a writer is ready, no new readers may start reading.

First Readers-Writers Problem Solution

- Data set
- Semaphore mutex initialized to 1
  - mutual exclusion for variable *readcount*.
- Semaphore wrt initialized to 1
  - for both reader and writer processes.
- Integer readcount initialized to 0.

```
while (true) {
        wait (wrt) ;
        // writing is performed
        signal (wrt) ;
}
```

Writer process

```
while (true) {
        wait (mutex) ;
                readcount ++ ;
                if (readcount == 1)
                        wait (wrt) ;
        signal (mutex)
        // reading is performed
        wait (mutex) ;
                readcount - - ;
                if (readcount  == 0)
                        signal (wrt) ;
        signal (mutex) ;
}
```

Reader Process

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)