Operating Systems
CMPSC 473
Synchronization
February 24, 2009 - Lecture 12
Instructor: Trent Jaeger
• Last class:
  – Synchronization Problems and Primitives
• Today:
  – Synchronization Solutions
More than just Exclusion

• But you also need synchronization constructs for other than exclusion.
  – E.g. If printer queue is full, I need to wait until there is at least 1 empty slot
  – Note that mutex_lock() / mutex_unlock() are not very suitable to implement such synchronization
  – We need constructs to enforce orderings (e.g. A should be done after B).
Semaphores

• You are given a data-type Semaphore_t.
• On a variable of this type, you are allowed
  – P(Semaphore_t) -- wait
  – V(Semaphore_t) – signal
• Intuitive Functionality:
  – Logically one could visualize the semaphore as having a counter initially set to 0.
  – When you do a P(), you decrement the count, and need to block if the count becomes negative.
  – When you do a V(), you increment the count and you wake up 1 process from its blocked queue if not null.
Semaphore Implementation

typedef struct {
    int value;
    struct process *L;
} semaphore_t;

void P(semaphore_t S) {
    S.value--;
    if (S.value < 0) {
        add this process to S.L and
        remove from ready queue
        context switch to another
    }
}

void V(semaphore_t S) {
    S.value++;
    if (S.value <= 0) {
        remove a process from S.L
        put it in ready queue
    }
}

NOTE: These are OS system calls, and there is no atomicity lost during
the execution of these routines (interrupts are disabled).
Binary vs. Counting Semaphores

• What we just discussed is a counting semaphore.
• A binary semaphore restricts the “value” field to just 0 or 1.
• We will mainly restrict ourselves to counting semaphores.

• Exercise: Implement counting semaphores using binary semaphores.
Semaphores can implement Mutex

```c
Semaphore_t m;

Mutex_lock() {
    P(m);
}

Mutex_unlock() {
    V(m);
}
```
Classic Synchronization Problems

- Bounded-buffer problem
- Readers-writers problem
- Dining Philosophers problem
- ....

- We will compose solutions using semaphores
Bounded Buffer problem

• A queue of finite size implemented as an array.
• You need mutual exclusion when adding/removing from the buffer to avoid race conditions.
• Also, you need to wait when appending to buffer when it is full or when removing from buffer when it is empty.
Bounded Buffer using Semaphores

int BB[N];
int count, head, tail = 0;
Semaphore_t m; // value initialized to 1
Semaphore_t empty; // value initialized to N
Semaphore_t full; // value initialized to 0

Append(int elem) {
    P(empty);
    P(m);
    BB[tail] = elem;
    tail = (tail + 1)%N;
    count = count + 1;
    V(m);
    V(full);
}

int Remove () {
    P(full);
    P(m);
    int temp = BB[head];
    head = (head + 1)%N;
    count = count - 1;
    V(m);
    V(empty);
    return(temp);
}
Readers-Writers Problem

• There is a database to which there are several readers and writers.

• The constraints to be enforced are:
  – When there is a reader accessing the database, there could be other readers concurrently accessing it.
  – However, when there is a writer accessing it, there cannot be any other reader or writer.
Readers-writers using Semaphores

Database db;
int nreaders = 0;
Semaphore_t m; // value initialized to 1
Semaphore_t wrt; // value initialized to 1

Reader() {
    P(m);
    nreaders++;
    if (nreaders == 1) P(wrt);
    .... Read db here ...
    P(m);
    nreaders--;
    if (nreaders == 0) V(wrt);
    V(m);
}

Writer() {
    P(wrt);
    .... Write db here ...
    V(wrt);
}
Dining Philosopher Problem

Philosophers alternate between thinking and eating.

When eating, they need both (left and right) chopsticks.

A philosopher can pick up only 1 chopstick at a time.

After eating, the philosopher puts down both chopsticks.
Semaphore_t chopstick[5];

Philosopher(i) {
  while () {
    P(chopstick[i]);
    P(chopstick[(i+1)%5];

    ... eat ...

    V(chopstick[i]);
    V(chopstick[(i+1)%5];

    ... think ...
  }
}

This is NOT correct!

Though no 2 philosophers use the same chopstick at any time, it can so happen that they all pick up 1 chopstick and wait indefinitely for another.

This is called a **deadlock**,
• Note that putting
  \[ P(chopstick[i]); \]
  \[ P(chopstick[(i+1)\%5]); \]
  within a critical section (using say \[ P(mutex)/V(mutex) \]) can avoid the deadlock.

• But then, only 1 philosopher can eat at any time!
int state[N];
Semaphore_t s[N]; // init. to 0
Semaphore_t mutex; // init. to 1

#define LEFT (i-1)%N
#define RIGHT (i+1)%N

philosopher(i) {
    while () {
        take_forks(i);
        eat();
        put_forks(i);
        think();
    }
}

take_forks(i) {
    P(mutex);
    state[i] = HUNGRY;
    test(i);
    V(mutex);
    P(s[i]);
}

put_forks(i) {
    P(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    V(mutex);
}

test(i) { /* can phil i eat? if so, signal that philosopher */
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        V(s[i]);
    }
}
Synchronization constructs

- Mutual exclusion locks
- Semaphores
- Monitors
- Critical Regions
- Path Expressions
- Serializers
- ....
Monitors

• An abstract data type consisting of
  – Shared data
  – Operations/procedures on this shared data

• External world only sees these operations (not the shared data or how the operations and sync. are implemented).

• Only 1 process can be “active” within the monitor at any time
  i.e. of all the processes that are executing monitor code, there can be at most 1 process in ready queue (rest are either blocked or not in monitor!)
• In addition, you have a condition variable construct available within a monitor.
  – Condition_t x, y;

• You can perform the following operations on a condition variable:
  – Wait(x): Process invoking this is blocked until someone does a signal.
  – Signal(x); Resumes exactly one blocked process.

• **NOTE:** If the signal comes before the wait, the signal gets lost!!! – You need to be careful since signals are not stored unlike semaphores.
• When P1 signals to wake up P2, note that both cannot be simultaneously running as per monitor definition.

• There are these choices:
  – Signalling process (P1) executes, and P2 waits until the monitor becomes free.
  – P2 resumes execution in monitor, while P1 waits for monitor to become free.
  – Some other process (waiting for entry) gets the monitor, while both P1 and P2 wait for monitor to become free.

• In general, try to write solutions that do not depend on which choice is used when implementing the monitor.
Structure of a Monitor

- Shared Data
  - Condition Variables
    - X
    - Y
  - Operations/Procedures
    - Append()
    - Remove()

- Initialization Code

Entry Queue
Bounded Buffer using Monitors

Monitor Bounded_Buffer:

Buffer[0..N-1];
int count= 0, head=tail=0;
Cond_t not_full, not_empty;

Remove() {
    if count == 0 wait(not_empty);
    Data = Buffer[tail];
    count--;
    tail = (tail+1)%N;
    if !empty(not_full) signal(not_full);
}

Append(Data) {
    if count == N   wait(not_full);
    Buffer[head] = Data
    count++;
    head = (head+1)%N;
    if !empty(not_empty) signal(not_empty);
}

Exercise

- Write monitor solutions for Readers-writers, and Dining Philosophers.
Pthreads Synchronization

• Mutex Locks
  – Protection Critical Sections
  – `pthread_mutex_lock(&lock), pthread_mutex_unlock(&lock)`
  – *What should we protect in project 2?*

• Condition Variables
  – For Value-based Control
  – `pthread_cond_wait(&cond), pthread_cond_signal(&cond)`
  – *Do we need condition vars for project 2?*
pthread_mutex_t lock;

big_lock() {

    pthread_mutex_init( &lock );
    
    /*
     ... initial code
    */
    pthread_mutex_lock( &lock );
    
    /*
     ... critical section
    */
    pthread_mutex_unlock( &lock );
    
    /*
     ... remainder
    */
}

Put code like around every critical section, like big_lock

What if reading and writing?
Readers-writers using Pthreads

thread_ongoing_t *ongoing;  // Initialization done elsewhere
int nr = 0, nw = 0;
pthread_cond_t OKR, OKW;

void req_read(void) {
    while (nw > 0) pthread_cond_wait(&OKR);
    nr++;
    pthread_cond_signal(&OKR);
}

void rel_read(void) {
    nr--;
    if (nr == 0) pthread_cond_signal(&OKW);
}

void req_write(void) {
    while (nr > 0 || nw > 0) pthread_cond_wait(&OKW);
    nw++;
}

void rel_write(void) {
    nw--;
    pthread_cond_signal(&OKW);
    pthread_cond_signal(&OKR);
}
Summary

• Semaphores
• Classical Synchronization Problems
• Monitors
• Implementation in Pthreads
• Next time: Deadlock