Classic Synchronization Problems

Lecture 15
March 9 2017
Classic Synchronization Problems

- Producer-Consumer
- Readers-Writer
- Dining Philosopher
- Sleeping Barber
Producer-Consumer Problem

- Bounded buffer: size $N$
- **Producer** process writes data to buffer
  - Writes to *In* and moves rightwards
- **Consumer** process reads data from buffer
  - Should not try to consume if there is no data
  - Reads from *Out* and moves rightwards
The consumer and producer should not access the same slot.

The consumer should not have to "busy wait" while waiting for the same slot.
PCP Preliminaries

- **Functions**
  - `produce_item()`: Producer calls this to generate something to put in buffer
    - Needs to determine next available slot
    - Needs to write `insert_item()`
  - `remove_item()`: Consumer calls to take item from buffer
  - `consume_item()`: Consumer calls this to do something with item
PCP Preliminaries

- **down, up**: semaphore operations
- Assume that the buffer has $N$ slots
- The variable `count` is used to keep track of the number of items in the buffer
- Shared entities: the buffer and the variable `count`, `In`, `Out`
This solution will make use of 2 semaphores that are initialized as follows:

- empty = N
- full = 0
PCP – Solution 1

Producer Code
void producer (void)
{
    int item;
    while (TRUE) {
        item = produce_item();
        down(empty);
        insert_item(item);
        up(full);
    }
}

Consumer Code
void consumer (void)
{
    int item;
    while (TRUE) {
        down(full);
        item = remove_item();
        up(empty);
        consume_item();
    }
}
- Say the consumer is scheduled first
- Since `full` is 0 the `down(full)` operation causes the consumer to block (`full = -1`)
PCP Solution 1

Consumer is scheduled out; Producer is now using the CPU

- Producer executes `down(empty)`, `empty = N-1`
- Producer puts in an item
PCP Solution 1

- empty = N-1
- full = -1

Producer executes `up(full)`

- full is now 0; and consumer is unblocked and ready to read
PCP Solution 1

- Consumer is allowed access to the buffer
- The value of full is 0 again.
PCP Solution 1

empty = N-1
full = 0

Consumer now executes \text{up}(empty)
PCP Solution 1

- Looks like we have a solution
- Tadah!
- Don’t celebrate yet!
  - What if you have multiple producers?
Consider the following scenario:
- Two producers apply the `down(empty)` operation
- One of the producers determines the next empty slot and is then scheduled out
- The second producer determines the next empty slot and gets the same result as the first producer
- Both producers write to the same slot

What now?
PCP - Solution 2

- Our next solution will make use of 3 semaphores that are initialized as follows:
  - mutex = 1
  - empty = N
  - full = 0
PCP - Solution 2

**Producer Code**

```c
void producer (void) {
    int item;
    while (TRUE) {
        item = produce_item();
        down(empty);
        down(mutex);
        insert_item(item);
        up(mutex);
        up(full);
    }
}
```

**Consumer Code**

```c
void consumer (void) {
    int item;
    while (TRUE) {
        down(full)
        down(mutex);
        item = remove_item();
        up(mutex);
        up(empty);
        consume_item();
    }
}
```
Say the consumer is scheduled first

Since full is 0 the down(full) operation causes the consumer to block. full us now -1
PCP Solution 2

- Now comes a producer
- The producer executes `down(empty)`
- It is allowed in. `empty` is now N-1
The producer executes `down(mutex); mutex = 0`

It is allowed to put something into the buffer

Can anyone else do something to the buffer?
- When the producer leaves it executes the \textbf{up}(mutex) operation,
- It then executes the \textbf{up}(full) operation, \textbf{full} 1
PCP Solution 2

- mutex = 1
- empty = N-1
- full = 1

- full is now 1 which is greater than 0
- Consumer is no longer blocked
- What happens to the value of full?
PCP Solution 2

- Consumer executes `down(full)` and `down(mutex)`
- Enters the critical section
- Consumes its item

```
mutex=1
empty = N-1
full = 1
```

```
mutex=0
empty = N-1
full = 0
```

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Out  In
Consumer executes `up(mutex)`

Consumer executes `up(empty)`
Readers-Writers Problem

- Many processes (readers) share a database
- Some processes (writers) write to the database
- Only one writer can be active at a time and must have mutual exclusive access
- Any number of readers can be active simultaneously
- This problem is non-preemptive
  - Wait for process in critical section to exit
- Models database access
Readers-Writers Problem

- Need to keep track of the number of readers current accessing data or wanting to access data
  - Let us denote this by \( r_c \)
- Writers have exclusive access to the shared data
Readers-Writers Problem

- There is a shared variable, $rc$, used to count the number of processes reading or wanting to read.
- There are two semaphores initialized as follows:
  - $mutex$ is initialized to 1
    - Used to control access to $rc$
  - $db$ is initialized to 1
    - Used to control access to database
void reader(void) {
    while (TRUE){
        down(&mutex);
        rc++;
        if (rc == 1) down(&db);
        up(&mutex);
        read_database();
        down (&mutex);
        rc--;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void write(void) {
    while (TRUE){
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}
Readers-Writers Problem

- Let us assume
  - 5 readers: R1, R2, R3, R4, R5
  - 3 writers: W1, W2, W3
- Initially no one is trying to use the database
Readers-Writers Problem

- R1 is the first to read the database
  - It finds \( rc \) is 0 and increments \( rc \) by 1
  - The condition \( rc == 1 \) is true
    - \( \text{down}(&db) \) is executed
    - R1 proceeds
  - Executes \( \text{up}(	ext{mutex}) \)
  - Gets access to the database
Readers-Writers Problem

- R2 comes along
  - Executes `down(mutex)`
  - Does it block?
    - No
  - What is the value of `rc` when it evaluates the condition `(rc == 1)`
    - `rc` is 2
  - The condition `rc == 1` is false
  - Executes `up(mutex)`
  - Gets access to the database
Readers-Writers Problem

- W1 comes along
  - Can W1 access the database?
    - No
  - Why?
    - What is the value of db? It is 0
    - W1 waits

- R1 is done reading the database
  - Decrements rc so that rc is now 1
  - W1 still can’t write
Readers-Writers Problem

- When can W1 write?
  - When R2 is done reading and thus it executes `up(db)`
- Now let’s say W1 is writing and that W2 wants to write
  - It gets blocked at `down(db)`
- R3 wants to read
  - It gets blocked at `down(db)`
Readers-Writers Problem

- W1 finishes
  - It executes `up(db)`
- Who goes first R3 or W2?
  - W2
- What if R3 arrived before W2 while W1 was executing? Who would go first?
  - R3
Readers-Writers Problem

- What if the order of arrival (while W1 is writing) is executing is R3, W2, R4?
  - R3, R4 would be able to read the database before W2.
- Doesn’t seem fair
- No. It turns out it is very difficult to be completely fair.
- One solution: Suspend readers that come after a writer
  - Reduces concurrency
Readers-Writers Problem

- **If there is a writer**
  - First reader blocks on
    - if (rc == 1) down(&db);
  - Other readers block on
    - mutex

- **Once a writer exits, all readers get to go through**

- **The last reader to exit signals a writer with**
  - if (rc == 0) up(&db)
Dining Philosopher’s Problem

- Formulated in 1965 by Edsger Dijkstra
  - Student exam exercise
- Used to model computers competing for access to tape drive peripherals.
- Used to model accesses to different data sets
- The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources
Dining Philosopher’s Problem

- 5 Philosophers around a table and 5 forks
- Philosophers eat (spaghetti)/think
- Eating needs 2 forks
- Pick one fork at a time (first the right fork then the left one and eat)
Dining Philosophers

- 5 Philosophers around a table and 5 forks
- Philosophers eat (spaghetti)/think
- Eating needs 2 forks
  - Spaghetti is very slippery
- Pick one fork at a time (first the right fork then the left one and eat)
The challenge is to grant requests for forks while avoiding deadlock and starvation.

Deadlock can occur if everyone tries to get their forks at once. Each gets a left fork, and is stuck, because each right fork is someone else’s left fork.
Dining Philosophers non-Solution

- The procedure `take_fork` waits until the specified fork is available and takes it.
- A philosopher takes the left fork and then the right fork.
- Suppose that all philosophers take the left fork at once?
- Not really serious

```c
void philosopher(int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat();
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```
Dining Philosophers Solution

#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

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

/* number of philosophers */
/* number of i’s left neighbor */
/* number of i’s right neighbor */
/* philosopher is thinking */
/* philosopher is trying to get forks */
/* philosopher is eating */
/* semaphores are a special kind of int */
/* array to keep track of everyone’s state */
/* mutual exclusion for critical regions */
/* one semaphore per philosopher */
/* i: philosopher number, from 0 to N-1 */
/* repeat forever */
/* philosopher is thinking */
/* acquire two forks or block */
/* yum-yum, spaghetti */
/* put both forks back on table */
void take_forks(int i) /* i: philosopher number, from 0 to N–1 */
{ 
    down(&mutex); /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i); /* try to acquire 2 forks */
    up(&mutex); /* exit critical region */
    down(&s[i]); /* block if forks were not acquired */
}

void put_forks(i) /* i: philosopher number, from 0 to N–1 */
{ 
    down(&mutex); /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT); /* see if left neighbor can now eat */
    test(RIGHT); /* see if right neighbor can now eat */
    up(&mutex); /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{ 
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
5 philosophers are initially in the thinking state (indicated by T in the State array)
A philosopher may move into eating state only if neither neighbour is eating
A philosopher blocks if the needed forks are busy
Dining Philosophers Solution

- Assume P0 wants to eat
- It calls `take_forks(0)`
- Records that it is hungry
Dining Philosophers Solution

- PO calls `test(0)` (within `take_forks(0)`)
- P1 and P4 are in thinking state
- PO can enter the eat state
- PO executes the `up(s[0])` operation in `test(0)`

<table>
<thead>
<tr>
<th>E</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>s</td>
</tr>
</tbody>
</table>
Dining Philosophers Solution

- PO executes the `down(s[0])` operation in `take_forks(0)`
- PO is now eating
Dining Philosophers Solution

- While P0 is eating, P1 decides it wants to eat
- It calls `take_forks (1)`
- Records that it is hungry
Dining Philosophers Solution

- P1 calls `test(1)` (within `take_forks(1)`)
- P0 is in an eating state; P2 is in a thinking state
- P1 is not successful in getting forks when it leaves `test(1)`
- P1 executes the `down(s[1])` operation in `take_forks(1)`
  - P1 is blocked
Dining Philosophers Solution

- What will unblock P1?
Dining Philosophers Solution

- PO finishes eating
- PO calls `put_forks()`
- PO is put into the thinking state
Dining Philosophers Solution

<table>
<thead>
<tr>
<th>T</th>
<th>E</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>s</td>
</tr>
</tbody>
</table>

- PO calls `test(P4)`
  - Does not change anything - P4 is not hungry
- PO calls `test(P1)`
  - Executes `up(s[1])`
    - P1 was blocked - it isn't any more so P1 can eat
Dining Philosophers Solution

- What if P0 and P2 were eating when P1 got hungry?
- The state of the arrays are above
Dining Philosophers Solution

- PO stops eating
- PO calls test(P1)
  - P1 cannot still cannot eat since state[RIGHT] indicates that P2 is eating.
- Have to wait until P2 calls test(P1)
Dining Philosopher’s Solution

How many philosophers can be eat at once?
The Sleeping Barber Problem

- N customer chairs (waiting chairs)
- One barber who can cut one customer’s hair at any time
- No waiting customer => barber sleeps
- Customer enters =>
  - If all waiting chairs full, customer leaves
  - Otherwise, if barber asleep, wake up barber and make him work
  - Otherwise, barber is busy - wait in a chair
The Sleeping Barber Solution

- **Global variable:** waiting
  - Use the **mutex** semaphore to control access

- **Signal to barber that it can sleep**
  - Use the **customer** semaphore

- **Signal to customer to wakeup if all barbers are busy**
  - Use the **barber** semaphore
# define CHAIRS 5  /* # chairs for waiting customers */
typedef int semaphore; /* use your imagination */
semaphore customers = 0;  /* # of customers waiting for service */
semaphore barbers = 0;  /* # of barbers waiting for customers */
semaphore mutex = 1;  /* for mutual exclusion */
int waiting = 0;  /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);  /* go to sleep if # of customers is 0 */
        down(&mutex);  /* acquire access to 'waiting' */
        waiting = waiting - 1;  /* decrement count of waiting customers */
        up(&barbers);  /* one barber is now ready to cut hair */
        up(&mutex);  /* release 'waiting' */
        cut_hair();  /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);  /* enter critical region */
    if (waiting < CHAIRS) {
        waiting = waiting + 1;  /* if there are no free chairs, leave */
        up(&customers);  /* increment count of waiting customers */
        up(&mutex);  /* wake up barber if necessary */
        up(&barbers);  /* release access to 'waiting' */
        go_to_sleep();  /* go to sleep if # of free barbers is 0 */
    } else {
        get_haircut();  /* be seated and serviced */
    }
}

Conclusions

- We have examined several non-trivial synchronization problems
  - Producer-Consumer Problem
  - Reader-Writer Problem
  - Dining Philosophers Problem
  - Sleeping Barber’s Problem