Concurrency and Synchronisation

Leonid Ryzhyk

Textbook

- Sections 2.3 & 2.5

Concurrency in operating systems

- Inter-process communication
- Intra-process communication
- Concurrency in the kernel

Concurrent vs sequential

- Sequential: program state depends on its previous state and the last instruction
- Concurrent: must take thread interleavings into account
Concurrent vs sequential

- Concurrent: must take thread interleavings into account

```c
void insert(item)
{
    item->next = head;
    head = item;
}
```

Race conditions

- Race condition: the result of the computation depends on the relative speed of two or more processes
  - Occur non-deterministically
  - Hard to debug

```c
void insert(struct node *item)
{
    item->next = head;
    head = item;
}
```

3 states, 2 transition, 1 execution trace

Question 1

- Question: How many states?

Race conditions

```c
void insert(struct node *item)
{
    item->next = head;
    head = item;
}
```

N processes

- Question: How many states?
Observation

- Unfortunately, it is usually easier to show something does not work, than it is to prove that it does work.
  - Ideally, we’d like to prove, or at least informally demonstrate, that our solutions work.

Dealing with race conditions

- **Approach 1: Mutual exclusion**
  - Identify shared variables
  - Identify code sections that access these variables (critical sections or critical regions)
  - Ensure that at most one process can enter a critical section

Dealing with race conditions

- **Approach 2: Lock-free data structures**
  - Allow concurrent access to shared variables, but make sure that they end up in a consistent state
  - Hard for non-trivial data structures
  - Performance overhead in the non-contended case

Dealing with race conditions

- **Approach 3: Message-based communication**
  - Eliminate shared variables
  - Processes communicate and synchronise using message passing

Mutual exclusion

- We can control access to the shared resource by controlling access to the code that accesses the resource
- Programming primitives:
  - `enter_region()` - called at the entrance to the critical region
  - `leave_region()` - called at the exit from the critical region
Mutual exclusion using critical regions

Example critical sections

```c
void insert(struct node *item) {
  enter_region(lock);
  item->next = head;
  head = item;
  leave_region(lock);
}

struct node *remove(void) {
  struct node *t;
  enter_region(lock);
  t = head;
  if (t->next != NULL) {
    head = head->next;
    leave_region(lock);
    return t;
  }
}
```

Implementing `enter_region` and `leave_region`

Requirements

- Mutual exclusion
  - No two processes simultaneously in the critical section
- No assumptions made about speeds of numbers of CPUs
- Liveness
  - No process must wait forever to enter the critical section

A solution?

- A lock variable
  - If lock == 1, `somebody is in the critical section and we must wait`
  - If lock == 0, `nobody is in the critical section and we are free to enter`

A solution?

```c
while(TRUE) {
  while(lock == 1);
  lock = 1;
  critical();
  lock = 0
  non_critical();
}
```

• Question: Any issues with this solution?
### Mutual exclusion by taking turns

```c
while(TRUE) {
    while (turn == 0);
    critical();
    turn = 1;
    non_critical();
}
```

- Works due to strict alternation
- Process must wait its turn even while the other process is doing something else.
  - Does not guarantee progress if a process no longer needs a turn.
  - Poor solution when processes require the critical section at differing rates

### Peterson's solution

```c
int turn;
int interested[2];

void enter_region(int process) {
    int other
    other = 1 - process;
    interested[process] = true;
    turn = process;
    while (turn == process && interested[other == TRUE]);
}

void leave_region(int process) {
    interested[process] = FALSE;
}
```

- Can be generalised to arbitrary number of processes
  - Run time is proportional to the maximal number of processes

### Hardware support for mutual exclusion

```c
while(TRUE) {
    while(lock == 0);
    lock = 1;
    critical();
    lock = 0
    non_critical();
}
```

- Test and set instruction
  - Writes 1 to a memory location and returns its old value as a single atomic operation
    - Atomic: As an indivisible unit (even on a multiprocessor).

### Mutual exclusion with test-and-set

```c
void enter_region(bool* lock)
{
    while(!test_and_set(lock) == 1);
}

void leave_region(bool* lock)
{
    *lock = 0;
}
```

### Other atomic instructions

- Compare-and-swap
  - Atomically compares the contents of a memory location to a given value and, if they are the same, modifies the contents of that memory location to a given new value.
- x86 supports atomic versions of most arithmetic instructions (using the lock prefix)

### Mutual exclusion for uniprocessors

- A uniprocessor system runs one thread at a time
- Concurrency arises from preemptive scheduling
- Question (recall of week 2): how does a thread switch occur?
Question 4

Mutual exclusion by disabling interrupts

- Before entering a critical region, disable interrupts
- After leaving the critical region, enable interrupts
- Pros
  - Simple
- Cons
  - Only available in the kernel
  - Blocks everybody else, even with no contention
    - Slows interrupt response time
  - Does not work on a multiprocessor

Tackling the busy-wait problem

- Most implementations of mutual exclusion discussed so far rely on busy waiting
  - A process sits in a tight loop waiting for the critical section to become available
  - Waste of CPU cycles and energy
- Sleep / Wakeup
  - Call sleep to block, instead of busy waiting
  - Another process calls wakeup to unblock the sleeping process

Tackling the busy-wait problem

```c
void enter_region(bool lock)
{
    if (test_and_set(lock) == 1)
        sleep();
}

void leave_region(bool lock)
{
    *lock = 0;
    wakeup();
}
```

- Question: What's wrong with this implementation?

Tackling the busy-wait problem

- Correct solution:

```c
typedef struct {
    bool locked; // queue of processes waiting for the mutex
    bool guard; // busy lock that protects access to the queue
} mutex;
```

Question 5

Hmm, you're using a lock to implement another lock. Isn't this a chicken and egg problem?

No, because we already know how to implement a busy lock.
Tackling the busy-wait problem

- Correct solution:

```c
void mutex_lock(mutex* lock) {
    enter_region(&lock->guard);
    add current process to lock->q
    mark current process as sleeping; //but keep it on the run queue
    leave_region(&lock->guard);
    schedule(); //move the process to the inactive queue
    } //if marked as sleeping
}

void mutex_unlock(mutex* lock) {
    enter_region(&lock->guard);
    wake the first process in lock->q
    leave_region(&lock->guard);
}
```

Mutual exclusion for user-level code

- Busy locks can be implemented at the user level, but are seldom used outside the kernel

- Blocking locks can only be implemented in the kernel and can be accessed from user-level processes via system calls.

Semaphores

- Semaphores, introduced by Dijkstra (1965), are a generalisation of mutual exclusion
  - A mutex allows at most one process to use a resource
  - A semaphore allows at most N processes

- Conceptually, a semaphore is a counter with two operations:
  - down() - atomically decrement the counter or block if the counter is 0
  - up() - atomically wake up one blocked process or increment the counter if there are no blocked processes

The producer-consumer problem

- Also called the bounded buffer problem
- A producer produces data items and stores the items in a buffer
- A consumer takes the items out of the buffer and consumes them.

Issues

- We must keep an accurate count of items in buffer
  - Producer
    - can sleep when the buffer is full
    - and wake up when there is empty space in the buffer
    - The consumer can call wakeup when it consumes the first entry of the full buffer
  - Consumer
    - Can sleep when the buffer is empty
    - And wake up when there are items available
    - Producer can call wakeup when it adds the first item to the buffer

- A semaphore with the counter initialised to one can be used as a mutex

- Implementation of semaphores is similar to the blocking mutex implementation
  - It uses a queue of waiting processes, a counter, and a busy lock used to protect the queue and the counter
  - Sleeping is implemented via calls to the OS scheduler
Pseudo-code for producer and consumer

```c
int main() {
    int count = 0;
    define N as buffer size
    prod() {
        while(TRUE) {
            if (count == 0)
                sleep();
            item = produce()
            if (count == N)
                remove_item();
            count--;
            insert_item();
            count++;
            if (count == 1)
                sleep();
        }
    }
    con() {
        while(TRUE) {
            if (count == 0)
                sleep();
            if (count == N)
                sleep();
            enter_region()
            insert_item();
            count--;
            leave_region();
            if (count == N-1)
                sleep();
            count++;
            enter_region()
            if (count == 1)
                sleep();
        }
    }
}
```

• Question: Any issues with this pseudo-code?

Proposed solution

• Let's use a mutex to protect the concurrent access

Proposed solution

```c
int main() {
    int count = 0;
    define N as buffer size
    prod() {
        while(TRUE) {
            if (count == 0)
                sleep();
            item = produce()
            if (count == N)
                remove_item();
            count--;
            insert_item();
            count++;
            if (count == 1)
                sleep();
        }
    }
    con() {
        while(TRUE) {
            if (count == 0)
                sleep();
            if (count == N)
                sleep();
            enter_region()
            insert_item();
            count--;
            leave_region();
            if (count == N-1)
                sleep();
            count++;
            enter_region()
            if (count == 1)
                sleep();
        }
    }
}
```

Problematic execution sequence

```c
prod() {
    while(TRUE) {
        if (count == 0)
            sleep();
        item = produce()
        if (count == N)
            remove_item();
        count--;
        insert_item();
        count++;
        if (count == 1)
            sleep();
    }
}
con() {
    while(TRUE) {
        if (count == 0)
            sleep();
        enter_region()
        insert_item();
        count--;
        leave_region();
        if (count == N-1)
            sleep();
        count++;
        leave_region();
        if (count == 1)
            sleep();
    }
}
```

...wakeup without a matching sleep is lost...

**Problem**

- The test for some condition and actually going to sleep needs to be atomic

- The following does not work
  ```
  enter_region()
  if (count == N)
  sleep();
  leave_region()
  ```

  The lock is held while asleep ⇒ count will never change

**Solving the producer-consumer problem with semaphores**

```c
#define N = 4
semaphore mutex = 1;
/
* count empty slots */
semaphore empty = N;
/
* count full slots */
semaphore full = 8;
```

**Solving the producer-consumer problem with semaphores**

```c
prod() {
    while(TRUE) {
        item = produce()
        down(empty);
        down(mutex);
        insert_item();
        up(mutex);
        up(full);
    }
}
```

```c
con() {
    while(TRUE) {
        down(full);
        down(mutex);
        remove_item();
        up(mutex);
        up(empty);
    }
}
```

**Summarising semaphores**

- Semaphores can be used to solve a variety of concurrency problems
- However, programming with them can be error-prone
  - Must up for every down for mutexes
    - Too many, or too few up's or down's, or up's and down's in the wrong order, can have catastrophic results
    - Must make sure that every use of a shared resource is protected by the semaphore

**Monitors**

- To ease concurrent programming, Hoare (1974) proposed monitors.
  - A higher level synchronisation primitive
  - Programming language construct
- Idea
  - A set of procedures, variables, data types are grouped in a special kind of module, a monitor
  - Only one process/thread can be in the monitor at any one time
  - Mutual exclusion is implemented by the compiler (which should be less error prone)

**Monitor**

- When a thread calls a monitor procedure that has a thread already inside, it is queued and it sleeps until the current thread exits the monitor.
Simple example

```java
monitor counter
{
    int count;
    procedure inc() {
        count = count + 1;
    }
    procedure dec() {
        count = count - 1;
    }
}
```

- Compiler guarantees only one thread can be active in the monitor at any one time
- Easy to see this provides mutual exclusion
  - No race condition on count.

Simple example

```java
monitor ProducerConsumer
{
    integer count;
    procedure insert(item: integer);
        begin
        if count = 0 then
            sleep;
        end
    procedure remove: integer;
        begin
        if count > 0 then
            sleep;
        end
    }
}
```

- Monitors provide more than just mutual exclusion
- Imagine that we want to implement a producer-consumer buffer as a monitor.

```java
procedure remove: integer;
    begin
    ...wakeup;
    ...
    end
end monitor
```

- sleeping inside the monitor prevents other threads from entering the monitor...

```
...hence wakeup will never be called
```

How do we block waiting for an event?

- We need a mechanism to block waiting for an event inside a monitor
- **Condition Variables**

Condition variables

- `wait()` releases the monitor lock, so that other processes can enter the monitor
- The lock is re-acquired before `wait()` returns
- To avoid race conditions, releasing the lock and blocking the process happen as one atomic operation

```
procedure x.wait();
```

- The operation `x.wait();` means that the process invoking this operation is suspended until another process invokes `x.signal();`
- The `x.signal()` operation resumes exactly one suspended process. If no process is suspended, then the `signal` operation has no effect.

```
...the monitor prevents other threads from entering the monitor...
...hence wakeup will never be called
```

Condition variables

- To allow a process to wait within the monitor, a **condition** variable must be declared, as

  ```java
  condition x, y;
  ```

- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation `x.wait();`
    means that the process invoking this operation is suspended until another process invokes `x.signal();`
  - The `x.signal()` operation resumes exactly one suspended process. If no process is suspended, then the `signal` operation has no effect.
Monitors

- Outline of producer-consumer problem with monitors

```c
monitor ProducerConsumer
condition full, empty;
integer count;
procedure insertion: integer;
begin
  if count = 0 then wait(full);  
  insert_item();
  count := count + 1;
  if count = full signal(empty);
end;

procedure consumer;
begin
  while true do
    item := ProducerConsumer.insert();
  end;
end;

function remove: integer;
begin
  if count = 0 then wait(empty);
  remove_item();
  count := count - 1;
  if count = 0 signal(empty);
end;

begin
  count := 0;
end;
end monitor;
```

OS/161 provided synchronisation primitives

- Locks
- Semaphores
- Condition Variables

Locks

- Functions to create and destroy locks
  ```c
  struct lock *lock_create(const char *name);
  void lock_destroy(struct lock *);
  ```

- Functions to acquire and release them
  ```c
  void lock_acquire(struct lock *);
  void lock_release(struct lock *);
  ```

Example use of locks

```c
int count;
struct lock *count_lock;
main()
  count = 0;
  count_lock = lock_create("count_lock");
  if (count_lock == NULL)
    panic("I'm dead");
  stuff();
}
```

Semaphores

```c
struct semaphore *sem_create(const char *name, int initial_count);
void sem_destroy(struct semaphore *);
void P(struct semaphore *);
void V(struct semaphore *);
```

Example use of semaphores

```c
int count;
struct semaphore *count_MUTEX;
main()
  count_mutex = sem_create("count_mutex");
  if (count_mutex == NULL)
    panic("I'm dead");
  stuff();
}
```

Condition variables

```c
struct cv *cv_create(const char *name);
void cv_destroy(struct cv *);
void cv_wait(struct cv *cv, struct lock *lock);
• Releases the lock and blocks
• Upon resumption, it re-acquires the lock
  – Note: we must reread the condition we slept on
void cv_signal(struct cv *cv, struct lock *lock);
void cv_broadcast(struct cv *cv, struct lock *lock);
• Wakes one/all, does not release the lock
• First “waker” scheduled after signaler releases the lock will re-acquire the lock
```

Note: All three variants must hold the lock passed in.

A producer-consumer solution using OS/161 CVs

```c
int count = 0;
define N 4 /* but size */
prod() {
    while(TRUE) {
        item = produce()
        lock_acquire()
        cv_wait[P1]
        insert_item(item);
        count++;
        if (count == n)
            cv_signal[e, i];
        lock_release()
    }
}
con() {
    while(TRUE) {
        lock_acquire()
        item = remove_item();
        count--;
        if (count == 0)
            cv_signal[f, i];
        lock_release()
    }
}
```

Dining philosophers

• Philosophers eat/think
• Eating needs 2 forks
• Pick one fork at a time
• How to prevent deadlock

![Dining philosophers diagram](image)

Solution to dining philosophers problem (part 1)

```c
#define N 5
#define LEFT 5
#define RIGHT 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
type semstate;
semaphore mutex = 1;
semaphore s[N];
void philosopher(int i) {
    while(TRUE) {
        think();
take_fork(i); % N;
eat();
put_fork(i);
put_fork((i+1) % N);
}
}
```

A non-solution to the dining philosophers problem

```c
#define N 5
void philosopher(int i) { /* i: philosopher number, from 0 to 4 */
    while(TRUE) { /* philosopher is thinking */
        take_fork(i); % take right fork, % is modulo operator +/
eat();
put_fork(i);
put_fork((i+1) % N);
    }
}
```

Solution to dining philosophers problem (part 2)
The readers and writers problem

- Models access to a database
  - E.g. airline reservation system
  - Can have more than one concurrent reader
- To check schedules and reservations
- Writers must have exclusive access
  - To book a ticket or update a schedule

A solution to the readers and writers problem

The sleeping barber problem

FYI

- Counting semaphores versus binary semaphores:
  - In a counting semaphore, count can take arbitrary integer values
  - In a binary semaphore, count can only be 0 or 1
  - Can be easier to implement
- Counting semaphores can be implemented in terms of binary semaphores (how?)

- Strong semaphores versus weak semaphores:
  - In a strong semaphore, the queue adheres to the FIFO policy
  - In a weak semaphore, any process may be taken from the queue
- Strong semaphores can be implemented in terms of weak semaphores (how?)