Concurrency and Synchronisation

Learning Outcomes

• Understand concurrency is an issue in operating systems and multithreaded applications
• Know the concept of a critical region.
• Understand how mutual exclusion of critical regions can be used to solve concurrency issues
  • Including how mutual exclusion can be implemented correctly and efficiently.
• Be able to identify and solve a producer consumer bounded buffer problem.
• Understand and apply standard synchronisation primitives to solve synchronisation problems.

Textbook

• Sections 2.3 - 2.3.7 & 2.5

Concurrency Example

count is a global variable shared between two threads.
After increment and decrement complete, what is the value of count?

```c
void increment ()
{
    int t;
    t = count;
    t = t + 1;
    count = t;
}
```

```c
void decrement ()
{
    int t;
    t = count;
    t = t - 1;
    count = t;
}
```

We have a race condition

Where is the concurrency?

• (a) Three processes each with one thread
• (b) One process with three threads

There is in-kernel concurrency even for single-threaded processes
Critical Region

• We can control access to the shared resource by controlling access to the code that accesses the resource.

⇒ A **critical region** is a region of code where shared resources are accessed.

• Variables, memory, files, etc...

• Uncoordinated entry to the critical region results in a race condition

⇒ Incorrect behaviour, deadlock, lost work...

Identifying critical regions

• Critical regions are regions of code that:
  • Access a shared resource,
  • and correctness relies on the shared resource not being concurrently modified by another thread/process/entity.

```c
void increment ()
{
    int t;
    t = count;
    t = t + 1;
    count = t;
}

void decrement ()
{
    int t;
    t = count;
    t = t - 1;
    count = t;
}
```

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

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

Example Race

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

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

Example critical regions

```c
struct node {
    int data;
    struct node *next;
};

struct node *head;

void init (void)
{
    head = NULL;
}

void insert (struct *item)
{
    item->next = head;
    head = item;
}
```
Critical Regions Solutions

- We seek a solution to coordinate access to critical regions.
- Also called critical sections
- Conditions required of any solution to the critical region problem
  1. Mutual Exclusion:
     • No two processes simultaneously in critical region
  2. No assumptions made about speeds or numbers of CPUs
  3. Progress
     • No process running outside its critical region may block another process
  4. Bounded
     • No process waits forever to enter its critical region

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?

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

A problematic execution sequence

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

Observation

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

Mutual Exclusion by Taking Turns

```
while (TRUE) {
  while (turn == 0) { /* loop */
    critical_region();
  }
  turn = 1;
  noncritical_region();
}
```

```
while (TRUE) {
  while (turn == 1) { /* loop */
    critical_region();
  }
  turn = 0;
  noncritical_region();
}
```

Proposed solution to critical region problem

(a) Process 0.
(b) Process 1.
Mutual Exclusion by Taking Turns

- **Works due to strict alternation**
- Each process takes turns
- **Cons**
  - Busy waiting
  - Process must wait its turn even while the other process is doing something else.
    - With many processes, must wait for everyone to have a turn.
    - Poor solution when processes require the critical section at differing rates.

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

Hardware Support for mutual exclusion

- **Test and set instruction**
  - Can be used to implement lock variables correctly
    - It loads the value of the lock
    - If lock \(= 0\),
      - set the lock to 1
      - return the result 0 — we acquire the lock
    - If lock \(= 1\)
      - return 1 — another thread/process has the lock
    - Hardware guarantees that the instruction executes atomically.
      - Atomically: As an indivisible unit.

Mutual Exclusion with Test-and-Set

- **Tackling the Busy-Wait Problem**

  - **Sleep / Wakeup**
    - The idea
    - When process is waiting for an event, it calls sleep to block, instead of busy waiting.
    - The event happens, the event generator (another process) calls wakeup to unblock the sleeping process.
    - Waking a ready/running process has no effect.

Test-and-Set

- **Pros**
  - Simple (easy to show it’s correct)
  - Available at user-level
  - To any number of processors
  - To implement any number of lock variables
- **Cons**
  - Busy waits (also termed a spin lock)
    - Consumes CPU
    - Starvation is possible when a process leaves its critical section and more than one process is waiting.
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
    • should 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
    • should 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

Pseudo-code for producer and consumer

```c
int count = 0;
#define N 4 /* buf size */

prod() {
  while(TRUE) {
    item = produce()
    if (count == N)
      sleep(prod);
    insert_item();
    count++;
    if (count == 1)
      wakeup(con);
  }
}

con() {
  while(TRUE) {
    if (count == 0)
      sleep(con);
    remove_item();
    count--;
    if (count == N-1)
      wakeup(prod);
  }
}
```

Problems

Concurrent uncontrolled access to the buffer

Concurrent uncontrolled access to the counter

Proposed Solution

• Lets use a locking primitive based on test-and-set to protect the concurrent access
Proposed solution?

```c
int count = 0;
#define N 4 /* buf size */
prod() {
    while(TRUE) {
        item = produce();
        if (count == N) {
            sleep(prod);
            acquire_lock();
            insert_item();
            count++;
            release_lock();
        }
        if (count == 1) {
            if (count >= N)
                sleep(prod);
            acquire_lock();
            insert_item();
            count++;
            release_lock();
        }
        if (count == N-1)
            if (count == 0)
                sleep(con);
            acquire_lock();
            remove_item();
            count--;
            release_lock();
            if (count == 0)
                wakeup(prod);}
    }
}
```

Problematic execution sequence

```c
con() {
    while(TRUE) {
        if (count == 0) {
            if (count == N)
                sleep(con);
            acquire_lock();
            remove_item();
            count--;
            release_lock();
            if (count == N-1)
                if (count == 0)
                    sleep(con);
                    acquire_lock();
                    remove_item();
                    count--;
                    release_lock();
                    if (count == 0)
                        wakeup(prod);}
    }
}
```

Problem

• The test for some condition and actually going to sleep needs to be atomic
• The following does not work:

```c
acquire_lock();
if (count == N)
    sleep();
release_lock();
```

The lock is held while asleep ⇒ count will never change

Semaphores

• Dijkstra (1965) introduced two primitives that are more powerful than simple sleep and wakeup alone.
  • P(): proberen, from Dutch to test.
  • V(): verhogen, from Dutch to increment.
  • Also called wait & signal, down & up.

How do they work

• If a resource is not available, the corresponding semaphore blocks any process waiting for the resource
• Blocked processes are put into a process queue maintained by the semaphore (avoids busy waiting!)
• When a process releases a resource, it signals this by means of the semaphore
• Signalling resumes a blocked process if there is any
• Wait (P) and signal (V) operations cannot be interrupted
• Complex coordination can be implemented by multiple semaphores

Semaphore Implementation

• Define a semaphore as a record
  ```c
typedef struct {
    int count;
    struct process *L;
  } semaphore;
  ```

• Assume two simple operations:
  • sleep suspends the process that invokes it.
  • wakeup(P) resumes the execution of a blocked process P.
Semaphore operations now defined as

**wait**:\nS.count--;
if (S.count < 0) {
    add this process to S.L;
sleep;
}

**signal**:\nS.count++;
if (S.count <= 0) {
    remove a process P from S.L;
wakeup(P);
}

- Each primitive is atomic
  - E.g. interrupts are disabled for each

---

### Semaphore as a General Synchronization Tool

- Execute B in P only after A executed in P,
- Use semaphore count initialized to 0
- Code:

```
P, P

<table>
<thead>
<tr>
<th>A</th>
<th>wait(flag)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>signal(flag)</td>
</tr>
</tbody>
</table>
```

---

### Semaphore Implementation of a Mutex

- Mutex is short for Mutual Exclusion
  - Can also be called a lock
  ```
  semaphore mutex;
  mutex.count = 1; /* initialise mutex */

  wait(mutex); /* enter the critical region */
  Blahblah();
  signal(mutex); /* exit the critical region */
  ```

Notice that the initial count determines how many waits can progress before blocking and requiring a signal ⇒ mutex.count initialised as 1

---

### Solving the producer-consumer problem with semaphores

```c
#define N = 4
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

prod() {
    while(TRUE) {
        item = produce();
        wait(empty);
        wait(mutex);
        insert_item();
        signal(mutex);
        signal(full);
    }
}

con() {
    while(TRUE) {
        wait(full);
        wait(mutex);
        remove_item();
        signal(mutex);
        signal(empty);
    }
}
```

---

### Solving the producer-consumer problem with semaphores

Summarising Semaphores

- Semaphores can be used to solve a variety of concurrency problems
- However, programming with them can be error-prone
  - E.g. must signal for every wait for mutexes
  - Too many, or too few signals or waits, or signals and waits in the wrong order, can have catastrophic results
Monitors

- To ease concurrent programming, Hoare (1974) proposed monitors.
  - A higher level synchronization primitive
  - Programming language construct

- Idea
  - A set of procedures, variables, data types are grouped in a special kind of module, a monitor.
  - Variables and data types only accessed from within the 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

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

Note: "paper" language

- 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.

Condition Variable

- To allow a process to wait within the monitor, a condition variable must be declared, as
  - 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
  - Another thread can enter the monitor while original is suspended
  - x.signal();
  - The signal operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

How do we block waiting for an event?

- We need a mechanism to block waiting for an event (in addition to ensuring mutual exclusion)
  - e.g., for producer consumer problem when buffer is empty or full
- Condition Variables
Condition Variables

OS/161 Provided Synchronisation Primitives

• Locks
• Semaphores
• Condition Variables

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();
}
```

procedure inc() {
    lock_acquire(count_lock);
    count = count + 1;
    lock_release(count_lock);
}

procedure dec() {
    lock_acquire(count_lock);
    count = count - 1;
    lock_release(count_lock);
}

Monitors

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

procedure remove(integer);
begin
    if count = N then wait(empty);
    count:= count - 1;
    if count > 0 then signal(full);
end;
```

• Outline of producer-consumer problem with monitors
  • only one monitor procedure active at one time
  • buffer has N slots

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 *);
```

Semaphores

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

```c
void P(struct semaphore *);
void V(struct semaphore *);
```
Example use of Semaphores

```c
int count;
struct semaphore *count_mutex;

main() {
  count = 0;
  count_mutex = sem_create("count", 1);
  if (count_mutex == NULL)
    panic("I'm dead");
  stuff();
}

procedure inc() {
  P(count_mutex);
  count = count + 1;
  V(count_mutex);
}

procedure dec() {
  P(count_mutex);
  count = count - 1;
  V(count_mutex);
}
```

Condition Variables

```c
struct cv *cv_create(const char *name);
void cv_destroy(struct cv *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 recheck 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 signaller releases the lock will re-acquire the lock

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

Condition Variables and Bounded Buffers

Non-solution

```c
lock_acquire(c_lock)
if (count == 0)
sleep();
remove_item();
count--;
lock_release(c_lock);
```

Solution

```c
lock_acquire(c_lock)
while (count == 0)
cv_wait(c_cv, c_lock);
remove_item();
count--;
lock_release(c_lock);
```

Alternative Producer-Consumer Solution Using OS/161 CVs

```c
int count = 0;
#define N 4 /* buf size */
prod() {
  item = produce();
  lock_acquire();
  while (count == N)
cv_wait(full, l);
  insert_item(item);
count++;
cv_signal(empty, l);
lock_release();
}
con() {
  lock_acquire();
  while (count == 0)
cv_wait(empty, l);
  item = remove_item();
count--;
cv_signal(full, l);
lock_release();
consume(item);
}
```

Dining Philosophers

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

```c
#define N 5
#define LEFT (N-1)%N
#define RIGHT (N+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define full 0
#define empty 1

typedef int semaphore;
int start(N); semaphore mutex = 1;

doctor phil[5];
void philosopher(int i) {
  
  while (TRUE) {
    think();
take_fork();
eat();
put_fork();
  }
}
```

Solution to dining philosophers problem (part 1)
Dining Philosophers

```c
#define N 5

void philosopher(int i) { /* i: philosopher number, from 0 to 4 */
    while (TRUE) { /* philosopher is thinking */
        think();
        take_fork(i);
        take_fork((i + 1) % N);
        eat();
        put_fork(i);
        put_fork((i + 1) % N);
    }
}
```

A nonsolution to the dining philosophers problem

---

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

```c
void reader(int id) { /* i: philosopher number, from 0 to N-1 */
    down(id); /* enter critical region */
    check(id); /* read the data */
    up(id); /* exit critical region */
}
```

A solution to the readers and writers problem

---

Solution to dining philosophers problem (part 2)