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 & 2.5

Making Single-Threaded Code Multithreaded

Conflicts between threads over the use of a global variable

Inter-Thread and Process Communication

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,…
Critical Regions

Also called critical sections

Conditions required of any solution to the critical region problem:

- Mutual Exclusion:
  - No two processes simultaneously in critical region
- No assumptions made about speeds or numbers of CPUs
- Progress
- Bounded
  - No process waits forever to enter its critical region

Example Race

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

Example critical sections

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

- Simple last-in-first-out queue implemented as a linked list.

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

Example critical sections

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

```c
struct node *remove(void)
{
  struct node *t;
  t = head;
  if (t != NULL) {
    head = head->next;
  }
  return t;
}
```
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.
  - Ideally, we'd like to prove, or at least informally demonstrate, that our solutions work.

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
    - 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

• See the textbook

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

Entering and leaving a critical region using the TSL instruction

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
    • Livelock in the presence of priorities
      – If a low priority process has the lock and a high priority process attempts to get it, the high priority process will busy-wait forever.
    • Starvation is possible when a process leaves its critical section and more than one process is waiting.

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 the event happens, the event generator (another process) calls wakeup to unblock the sleeping process.
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.

Pseudo-code for producer and consumer

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

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

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

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 /* buf size */
prod() {
while(TRUE) {
      item = produce();
      if (count == N)
            sleep();
      acquire_lock();
      insert_item();
      count++;
      release_lock();
      if (count == 1)
            wakeup(con);
      }
}  
}
```

Problematic execution sequence

```c
con() {
while(TRUE) {
      if (count == 0)
            sleep();
      acquire_lock();
      remove_item();
      count--;
      release_lock();
      if (count == N-1)
            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 and signal operations cannot be interrupted
- Complex coordination can be implemented by multiple semaphores

Semaphore Implementation

```c
typedef struct {
      int count;
      struct process *L;
} semaphore;
```

- Define a semaphore as a record
- 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

\[
\text{wait}(S):
\]

\[
S.\text{count}--;
\]

if (S.count < 0) {
    add this process to S.L;
    sleep;
}

\[
\text{signal}(S):
\]

\[
S.\text{count}++;
\]

if (S.count <= 0) {
    remove a process P from S.L;
    wakeup(P);
}

• Each primitive is atomic

Semaphore as a General Synchronization Tool

• Execute B in \( P_j \) only after A executed in \( P_i \)
• Use semaphore count initialized to 0
• Code:

\[
\begin{align*}
P_i & \quad P_j \\
\vdots & \quad \vdots \\
A & \quad \text{wait(flag)} \\
\text{signal(flag)} & \quad B
\end{align*}
\]

Semaphore Implementation of a Mutex

• Mutex is short for Mutual Exclusion
  – Can also be called a lock

\[
\text{mutex count = 1; /* initialise mutex */}
\]

\[
\text{wait(mutex); /* enter the critical region */}
\]

\[
\text{Blahblah();}
\]

\[
\text{signal(mutex); /* exit the critical region */}
\]

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

\[
\begin{align*}
\#define N = 4 \\
\text{semaphore mutex = 1;}
\]

\[
\text{/* count empty slots */}
\]

\[
\text{semaphore empty = N;}
\]

\[
\text{/* count full slots */}
\]

\[
\text{semaphore full = 0;}
\]

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);
    }
}

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 synchronisation 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 example
    integer c;
    condition x;

    procedure producer();
        ...
    end;

    procedure consumer();
        ...
    end;
end monitor:

Example of a monitor
```

Condition Variable

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

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

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

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);
}
```

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

```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 = 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(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

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);
```

A Producer-Consumer Solution Using OS/161 CVs

```c
int count = 0;
define N 4 /* buf size */
prod() {
    while(TRUE) {
        item = produce();
        lock_acquire(l)
        while (count == N)
            cv_wait(f,l);
        insert_item(item);
        count++;
        if (count == 1)
            cv_signal(e,l);
        lock_release();
    }
}
con() {
    while(TRUE) {
        lock_acquire(l)
        while (count == 0)
            cv_wait(e,l);
        item = remove_item();
        count--;
        if (count == N-1)
            cv_signal(f,l);
        lock_release(l);
        consume(item);
    }
}
```

Dining Philosophers

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

Solution to dining philosophers problem (part 1)
Dining Philosophers

```c
#define N 5

void philosopher(int i) {
    while (TRUE) { /* philosopher is thinking */
        think(i);
        take_fork(i);
        take_fork((i + 1) % N);
        eat();
        put_fork(i);
        put_fork((i + 1) % N);
    }
}

A nonsolution to the dining philosophers problem
```

Solution to dining philosophers problem (part 2)

```c

Solution to 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

The Readers and Writers Problem
```

A solution to the readers and writers problem