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

We have a race condition

Two processes want to access shared memory at the same time

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,…
Mutual exclusion using critical regions

Example critical sections

```c
struct node *remove(void) {
    struct node *t;
    if (t != NULL) {
        return t;
    }
}
```

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
  - No process running outside its critical region may block another process
- 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?

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

A problematic execution sequence

```c
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

Proposed solution to critical region problem

(a) Process 0. (b) Process 1.
**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.
  - Atomics: As an indivisible unit.

---

**Mutual Exclusion with Test-and-Set**

- **Instruction**
  - `test_and_set lock`
    - copy lock to register and set lock to 1
    - `cmp register, 0`
    - if lock was 0, lock was set, so loop
    - `jne critical_region`
    - `mov 1 lock, 0` — store a 0 in lock
    - `sla 1, return to caller`

- **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.

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 wake up 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 wake up when it adds the first item to the buffer

Pseudo-code for producer and consumer
```
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);
  }
}
```

Problems
• Concurrent uncontrolled access to the buffer
• Concurrent uncontrolled access to the counter

Proposed Solution
• Let's 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);
    }
}
con() {
    while(TRUE) {
        if (count == 0) sleep();
        acquire_lock();
        remove_item();
        count--;
        release_lock();
        if (count == N-1) wakeup(prod);
    }
}
```

Problematic execution sequence

```c
cond() {
    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

- 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 as a General Synchronization Tool

- Execute $B$ in $P_j$ only after $A$ executed in $P_i$.
- Use semaphore $count$ initialized to 0.
- Code:

\[
\begin{array}{cccc}
  P_i & P_j \\
  \vdots & \vdots \\
  A & \text{wait(flag)} \\
  \text{signal(flag)} & B
\end{array}
\]

Semaphore Implementation of a Mutex

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

```c
semaphore mutex;
mutex.count = 1; /* initialize mutex */
wait(mutex); /* enter the critical region */
blablablablablablabla();
signal(mutex); /* exit the critical region */
```

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

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 = 0;
```

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

```c
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)

Simple example

- The operation has no effect.
- Condition variables
  - 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.

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 Variable

- To allow a process to wait within the monitor, a condition variable must be declared, as
  \[ \text{condition } x, y; \]
- Condition variable can only be used with the operations \text{wait} and \text{signal}.
  - The operation \text{x.wait()}; means that the process invoking this operation is suspended until another process invokes \text{x.signal()};
  - The \text{x.signal} operation resumes exactly one suspended process. If no process is suspended, then the \text{signal} operation has no effect.
Condition Variables

OS/161 Provided Synchronisation Primitives

- Locks
- Semaphores
- Condition Variables

Example use of locks

```
int count;
struct lock *count_lock

main() {
    count = 0;
    count_lock = lock_create("count_lock");
    if (count_lock == NULL)
        panic("I'm dead");
    stuff();
}
```

Functions to create and destroy locks

```
struct lock *lock_create(const char *name);
void    lock_destroy(struct lock *);
```

Functions to acquire and release them

```
void    lock_acquire(struct lock *);
void    lock_release(struct lock *);
```

Example use of locks

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

Semaphores

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

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 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 "water" 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
lock_acquire(c_lock)
if (count == 0)
sleep();
remove_item();
count--;
lock_release(c_lock);

### Solution
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 /* buf size */
prod() { 
    while(TRUE) {
        item = produce()
        lock_acquire()
        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()
        while (count == 0)
            cv_wait(e, l);
        item = remove_item();
        count--;
        if (count == N-1)
            cv_signal(f, l);
        lock_release();
    }
}
```

Dining Philosophers

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

```c
#define N 4 /* buf size */
#define M 5 /* max count */

prod() { 
    while(TRUE) {
        item = produce()
        lock_acquire()
        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()
        while (count == 0)
            cv_wait(e, l);
        item = remove_item();
        count--;
        if (count == N-1)
            cv_signal(f, l);
        lock_release();
    }
}
```

Solution to dining philosophers problem (part 1)
Dining Philosophers

```c
#define N 5

void philosopher(int i) {  
  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
```

### Solution to dining philosophers problem (part 2)

```c
void eating(int n, int i) {  
  printf("Philosopher %d is eating...
", i);

  //philosopher is eating  
  eat();  

  //philosopher is thinking  
  think();  

  //philosopher puts fork on the table  
  put_fork(i);
}
```

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
// A solution to the readers and writers problem
```