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

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

Critical Regions

Example critical sections

Example critical sections

Critical Regions

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 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.

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

Mutual Exclusion by Taking Turns

```
while (i < N) {
    while (turn != i) / loop */
    critical_region();
    turn = i;
    non_critical_region();
}
```

Proposed solution to critical region problem
(a) Process 0. (b) Process 1.

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

```
while (i < N) {
    while (turn != i) / loop */
    critical_region();
    turn = i;
    non_critical_region();
}
```

Peterson’s Solution

- See the textbook
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
  - 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

- enter_region:
  - CMP HL,ROD (LH, ROD) | copy lock to register and set lock to 1
  - JNL enter_region | if lock was not zero, lock was set, so loop
  - HALT | return to caller, critical region continues

- leave region:
  - MOV E, RO | start a 0 in lock
  - HLT | 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
    - Live-lock 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 wakeup 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
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);
  }
}
```

Problems

```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);
  }
}
```

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();
    acquire_lock();
    remove_item();
    count--;
    release_lock();
    if (count == N-1) wakeup(prod);
  }
}

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

Problematic execution sequence

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

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

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

```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:

```c
P_i

... ...

A   wait(flag)

signal(flag)  B
```
Semaphore Implementation of a Mutex

- Mutex is short for Mutual Exclusion
  - Can also be called a lock
  ```c
  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 \( \Rightarrow \) 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;

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)
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:
  ```
  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
  - only one monitor procedure active at one time
  - buffer has flip-flops

Example of a monitor

```
monitor example
integer r;
condition c:

procedure producer()
...
end;

procedure consumer()
...
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();
}
```

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);
}

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(const char *name);
void cv_destroy(struct cv *);
void cv_wait(struct cv *cv, struct lock *lock);
  // Re-acquires 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 signaler releases the lock will re-acquire the lock
```

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

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

Dining Philosophers - Solution

```c
#define N 5 /* number of philosophers */
#define M (N-1) /* number of left neighbors */
#define R (N-1) /* number of right neighbors */
#define L (N-1) /* number of left forks */

struct philosopher {
    int id; /* philosopher number, from 0 to N-1 */
    int思维; /* philosopher is thinking */
    int left_fork; /* left fork */
    int right_fork; /* right fork */
    int is_eating; /* is eating */
    int is_thinking; /* is thinking */
    int is_buffers; /* is buffers */
    int is_table; /* is table */
};

void philosopher(int id) {
    /* philosopher number, from 0 to N-1 */
    int left_fork(id % N); /* left fork */
    int right_fork((id+1) % N); /* right fork */
    int is_eating(id % 2); /* is eating */
    int is_thinking(id % 2); /* is thinking */
    int is_buffers(id % 2); /* is buffers */
    int is_table(id % 2); /* is table */
}
```

A nonsolution to the dining philosophers problem

Dining Philosophers - Solution (part 1)

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

Solution

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

Solution

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

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);
```

Dining Philosophers - Solution (part 2)

```c
void philosopher(int id) {
    /* philosopher number, from 0 to N-1 */
    int left_fork(id % N); /* left fork */
    int right_fork((id+1) % N); /* right fork */
    int is_eating(id % 2); /* is eating */
    int is_thinking(id % 2); /* is thinking */
    int is_buffers(id % 2); /* is buffers */
    int is_table(id % 2); /* is table */
}
```
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