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 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,…
Critical Regions

Mutual exclusion using critical regions
Example critical sections

```
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;
        head = head->next;
        return t;
    }
    return NULL;
}

• Simple last-in-first-out queue implemented as a linked list.
```
Example Race

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

```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;
}
```
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**
  - 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?

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

```
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();
}

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

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

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

(a) Process 0.  (b) Process 1.

Proposed solution to critical region problem
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

**enter** _region:_

- TSL REGISTER,LOCK  | copy lock to register and set lock to 1
- CMP REGISTER,#0    | was lock zero?
- JNE enter_region    | if it was non zero, lock was set, so loop
- RET | return to caller; critical region entered

**leave** _region:_

- MOVE LOCK,#0        | store a 0 in lock
- RET | 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 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
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();
        remove_item();
        count--;
        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
Problems

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

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

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

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

    if (count == N)
        sleep();
    acquire_lock()
    remove_item();
    count--;
    release_lock();
    if (count == N-1)
        wakeup(prod);
}
}
\end{verbatim}

```
con() {
    while(TRUE) {
        if (count == 0)
            wakeup without a matching sleep is lost
        insert_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 operations now defined as

\[ \text{wait}(S): \]
\[
S.\text{count}--; \\
\text{if} \ (S.\text{count} < 0) \ { \\
\quad \text{add this process to } S.L; \\
\quad \text{sleep;} \\
\}
\]

\[ \text{signal}(S): \]
\[
S.\text{count}++; \\
\text{if} \ (S.\text{count} \leq 0) \ { \\
\quad \text{remove a process } P \text{ from } S.L; \\
\quad \text{wakeup}(P); \\
\}
\]

• 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:

$$
\begin{align*}
  & P_i & P_j \\
  & : & : \\
  & A & wait(flag) \\
  & signal(flag) & B
\end{align*}
$$
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 ⇒ mutex.count initialised as 1
Solving the producer-consumer problem with semaphores

#define N = 4

semaphore mutex = 1;

/* count empty slots */
semaphore empty = N;

/* count full slots */
semaphore full = 0;
Solving the producer-consumer problem with semaphores

```c
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.
Monitors

```plaintext
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
    end;

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

Example of a monitor
Simple example

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.
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
monitors

monitor ProducerConsumer
condition full, empty;
integer count;
procedure insert(item: integer);
begin
  if count = N then wait(full);
  insert_item(item);
  count := count + 1;
  if count = 1 then signal(empty)
end;
function remove: integer;
begin
  if count = 0 then wait(empty);
  remove = remove_item;
  count := count - 1;
  if count = N - 1 then signal(full)
end;
count := 0;
end monitor;

procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
  end
end
procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
  end
end;

• 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
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);
  // 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 “waiter” 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
Dining Philosophers

#define N 5 /* number of philosophers */
#define LEFT (i+N-1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i) /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {
        think(); /* repeat forever */
        take_forks(i); /* philosopher is thinking */
        eat(); /* acquire two forks or block */
        put_forks(i); /* yum-yum, spaghetti */
        put both forks back on table */
    }
}
Dining Philosophers

#define N 5

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

/* number of philosophers */
/* i: philosopher number, from 0 to 4 */
/* philosopher is thinking */
/* take left fork */
/* take right fork; % is modulo operator */
/* yum-yum, spaghetti */
/* put left fork back on the table */
/* put right fork back on the table */

A nonsolution to the dining philosophers problem
Dining Philosophers

```c
void take_forks(int i) {
    down(&mutex);  /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i); /* try to acquire 2 forks */
    up(&mutex); /* exit critical region */
    down(&s[i]); /* block if forks were not acquired */
}

void put_forks(i) {
    down(&mutex); /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT); /* see if left neighbor can now eat */
    test(RIGHT); /* see if right neighbor can now eat */
    up(&mutex); /* exit critical region */
}

void test(i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

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
The Readers and Writers Problem

typedef int semaphore; /* use your imagination */
semaphore mutex = 1; /* controls access to 'rc' */
semaphore db = 1; /* controls access to the database */
int rc = 0; /* # of processes reading or wanting to */

void reader( void )
{
    while (TRUE) { /* repeat forever */
        down( &mutex ); /* get exclusive access to 'rc' */
        rc = rc + 1; /* one reader more now */
        if (rc == 1) down( &db ); /* if this is the first reader ... */
        up( &mutex ); /* access the data */
        read_data_base(); /* get exclusive access to 'rc' */
        down( &mutex ); /* one reader fewer now */
        rc = rc - 1;
        if (rc == 0) up( &db ); /* if this is the last reader ... */
        up( &mutex ); /* release exclusive access to 'rc' */
        use_data_read(); /* noncritical region */
    }
}

void writer( void )
{
    while (TRUE) { /* repeat forever */
        think_up_data(); /* noncritical region */
        down( &db ); /* get exclusive access */
        write_data_base(); /* update the data */
        up( &db ); /* release exclusive access */
    }
}