Synchronisation and Concurrency II

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
- Example of a monitor

Simple example
- 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.

Example of a monitor
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 wait and signal.
  - The operation \[x \text{.wait();}\] means that the process invoking this operation is suspended until another process invokes \[x \text{.signal();}\].
  - The \[x \text{.signal();}\] operation resumes exactly one suspended process. If no process is suspended, then the \text{signal} operation has no effect.

**Monitors**

- 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
  
  \[
  \text{struct lock *lock_create(const char *name);} \\
  \text{void lock_destroy(struct lock *);} \\
  \]

- Functions to acquire and release them
  
  \[
  \text{void lock_acquire(struct lock *);} \\
  \text{void lock_release(struct lock *);} \\
  \]
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);
}

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 *, struct lock *);
void cv_signal(struct cv *, struct lock *);
void cv_broadcast(struct cv *, struct lock *);
```

- Releases the lock and blocks
- Upon resumption, it re-acquires the lock
  - Note: we must re-check the condition we slept on
- 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();
    }
}
```

```c
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);
        consume(item);
        lock_release();
    }
}
```
Interprocess Communication

- **Shared Memory**
  - Region of memory appears in each process
  - Communication via modifications to shared region
  - Requires concurrency control (semaphores, mutexes, monitors…)

- **Shared files**
  - Cumbersome

- **Message Passing**
  - "real" IPC
  - Requires two facilities
    - send(message)
      - Message may be fixed or variable in size
    - receive(message)
  - OS ships the data from the sender to the receiver

IPC design issues

- Is the communication synchronous or asynchronous?
- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is the message format fixed or variable?
- Is a link unidirectional or bi-directional?

Blocking vs. Non-blocking

- **Send**
  - Operation blocks until partner is ready to receive
    - Rendezvous model
    - Send and receiver execute their system at the same time (synchronously)
  - Asynchronous

- **Receive**
  - Operation blocks until message is available
    - synchronous
  - Asynchronous (polling)
Blocking vs. Non-blocking

- **Non-blocking IPC**
  - Requires buffering of messages in the kernel
  - May fail due to buffer full
  - Overhead (copying, allocation)
  - Higher level of concurrency
  - Requires a separate synchronisation primitive

- **Blocking IPC**
  - May lead to threads blocked indefinitely
  - Can use *timeouts* prevent this
  - Zero-timeout $\Rightarrow$ non-blocking receive

Direct Communication

- Processes (or threads) must name each other explicitly using their unique process (or thread) ID:
  - $\text{send}(P, \text{message})$ – send a message to process $P$
  - $\text{receive}(Q, \text{message})$ – receive a message from process $Q$

- Properties of communication link
  - Links are established automatically (implicitly).
  - A link is associated with exactly one pair of communicating processes.
  - Between each pair there exists exactly one link.
  - The link may be unidirectional, but is usually bi-directional.

Indirect Communication

- Messages are directed to and received from mailboxes (also referred to as ports).
  - Each mailbox has a unique id.
  - Processes can communicate only if they share a mailbox.
  - E.g. Mach

- Properties of communication link
  - Link established only if processes share a common mailbox
  - OS mechanism required to establish mailbox sharing
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link may be unidirectional or bi-directional.

Indirect Communication

- Mailbox sharing
  - $P_1$, $P_2$, and $P_3$ share mailbox A.
  - $P_1$ sends; $P_2$ and $P_3$ receive.
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation (Mach).
  - Allow the system to select arbitrarily the receiver.
  - First come, first served.

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:
  - $\text{send}(A, \text{message})$ – send a message to mailbox $A$
  - $\text{receive}(A, \text{message})$ – receive a message from mailbox $A$

Message Passing

```
Message in list

void produce(void)
{
    int item;
    message n;
    while (true) {
        item = produce_item();
        sendproducer( item);
        send(item, message);
        send(item, message);
        send(item, message);
        send(item, message);
    }
}
```

```
Message out

void consume(void)
{
    int item;
    message n;
    while (true) {
        item = receive_item();
        message "hello"
        message "world"
        message "hi"
        message "bye"
        send(item, message);
    }
}
```

The producer-consumer problem with $N$ messages
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)

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

A solution to the readers and writers problem
The Sleeping Barber Problem

See the textbook

Solution to sleeping barber problem.