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.

Example of a monitor

```
monitor example
integer e;
condition c;
procedure producer();
...
end;
procedure consumer();
...
end;
end 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

- 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 $N$ slots

OS/161 Provided Synchronisation Primitives

- Locks
- Semaphores
- Condition Variables

Locks

- 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

```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();
}
```

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

```c
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();
}
```

```c
procedure inc() {
    P(count_mutex);
    count = count + 1;
    V(count_mutex);
}
```

```c
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 *cv);
void cv_wait(struct cv *cv, struct lock *lock); // Releases the lock and blocks
void cv_signal(struct cv *cv, struct lock *lock); // Wakes one/all, does not release the lock
void cv_broadcast(struct cv *cv, struct lock *lock); // Wakes one/all, does not release 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()
        while (count == N)
            cv_wait(f, l);
        insert_item(item);
        count++;
        if (count == 1)
            cv_signal(e, l);
        lock_release()
    }
}
```

```c
con() {
    while(TRUE) {
        if (count == N-1)
            cv_signal(f, l);
        lock_acquire()
        while (count == 0)
            cv_wait(e, l);
        item = remove_item();
        count--;
        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

**Interprocess Communication (IPC)**

- Mechanism for processes to communicate and synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive

**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 system at the same time
      - (synchronously)
  - Asynchronous

- **Receive**
  - Operation blocks until message is available
  - synchronous

- **Send**
  - Kernel receives message and delivers when receiver is ready
  - Asynchronous

- **Receive**
  - System call returns immediately if no message is available
  - 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 ⇒ non-blocking receive

Direct Communication

- Processes (or threads) must name each other explicitly using their unique process (or thread) ID:
  - `send(P, message)` – send a message to process P
  - `receive(Q, 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

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

- Primitives are defined as:
  - `send(A, message)` – send a message to mailbox A
  - `receive(A, message)` – receive a message from mailbox A

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.

Message Passing

```c
#include <stdio.h>

#define BUFFER_SIZE 100

int main(int argc, char *argv[])
{
    FILE *fp;
    char buffer[BUFFER_SIZE];

    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        printf("Error opening file \n");
        return 1;
    }

    fprintf(buffer, "Hello, world! \n");
    fwrite(buffer, strlen(buffer), 1, fp);

    fclose(fp);
    return 0;
}
```

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)

Dining Philosophers

define N 5

define M 1

#define HUNGRY 1
#define THINKING 0
#define LEFT (n-1)\n#define RIGHT n

define (EATING) 2

typed int semaphore;
semaphore mutex = 1;
semaphore right;

void philosopher(int i)
{
    while (TRUE)
    {
        take_fork(i);
        eat();
        put_fork(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

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

Solution to sleeping barber problem.