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

```
monitor example
  integer i;
  condition c;

  procedure producer( );
    .
    .
    .
    end;

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

Example of a monitor
Simple example

```c
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

  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.
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;

procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    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

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

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

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
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 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);
    }
}
```
Interprocess Communication

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

![Diagram showing shared data between Process 1 and Process 2]
Interprocess Communication

• Shared files
  – Cumbersome
Interprocess Communication

- 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 to *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 their system at the same time (synchronously)

• 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:
  – \texttt{send}(P, message) – send a message to process P
  – \texttt{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:
  \textbf{send}(A, \textit{message}) – send a message to mailbox A
  \textbf{receive}(A, \textit{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
#define N 100 /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m; /* message buffer */

    while (TRUE) {
        item = produce_item(); /* generate something to put in buffer */
        receive(consumer, &m); /* wait for an empty to arrive */
        build_message(&m, item); /* construct a message to send */
        send(consumer, &m); /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m); /* get message containing item */
        item = extract_item(&m); /* extract item from message */
        send(producer, &m); /* send back empty reply */
        consume_item(item); /* do something with the item */
    }
}
```

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
#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
#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) {
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

Solution to dining philosophers problem (part 1)
Dining Philosophers

void take_forks(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

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

```c
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); /* release exclusive access to 'rc' */
        read_data_base(); /* access the data */
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc - 1; /* one reader fewer now */
        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 */
    }
}
```

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

Solution to sleeping barber problem.

```c
#define CHAIRS 5 /* # chairs for waiting customers */
typedef int semaphore; /* use your imagination */
semaphore customers = 0; /* # of customers waiting for service */
semaphore barbers = 0; /* # of barbers waiting for customers */
semaphore mutex = 1; /* for mutual exclusion */
int waiting = 0; /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers); /* go to sleep if # of customers is 0 */
        down(&mutex); /* acquire access to 'waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&barbers); /* one barber is now ready to cut hair */
        up(&mutex); /* release 'waiting' */
        cut_hair(); /* cut hair (outside critical region) */
    }
}

void cut_hair(void)
{
    down(&mutex); /* enter critical region */
    if (waiting < CHAIRS) {
        waiting = waiting + 1; /* increment count of waiting customers */
        up(&customers); /* wake up barber if necessary */
        up(&mutex); /* release access to 'waiting' */
        down(&barbers); /* go to sleep if # of free barbers is 0 */
        get_haircut(); /* be seated and be serviced */
    } else {
        up(&mutex); /* shop is full; do not wait */
    }
}
```