Task Synchronisation
2501ICT/7421ICT
Nathan

René Hexel
School of Information and Communication Technology
Griffith University

Semester 1, 2012
Outline

1. Multitasking Review
   - Threads and Processes Reviewed

2. Concurrency and Synchronisation
   - Concurrency
   - Task Synchronisation
   - Typical Problems

3. Deadlock and Starvation
   - Deadlocks
   - Strategies
   - Dining Philosophers
Multitasking Review

- Multitasking allows programs to do more than one thing at a time
  - Multiprocessing: multiple CPUs
  - Timesharing: single CPU

- Processes vs. Threads
  - Processes: memory protection, heavyweight
  - Threads: no protection, lightweight

- Scheduling and Dispatching
  - Scheduler: high-level queuing algorithms
  - Dispatcher: low-level CPU assignment
Concurrency Problems

- Two Tasks accessing common resources (e.g. memory)
  - no problem as long as both tasks only read
  - what happens if one task writes while the other task reads?
  - what happens if both tasks try writing?
- Let’s look at some examples!
Example (two tasks modifying shared data)

```c
int shared = 0;

void task1(void)
{
    shared = 1;
}

task2:

extern int shared;

void task2(void)
{
    shared = 2;
}
```

- No concurrency problem!
  - `shared` is either 0, 1, or 2
- Both tasks use Atomic Operations
Example (two tasks modifying shared data)

```c
int shared = 0;

void task1(void)
{
    shared++;
    shared++;
}

void task2(void)
{
    shared += 2;
}

extern int shared;
```

- Inconsistencies can occur!
  - tasks can interrupt each other at critical points
  - *Read-Modify-Write* operations are not Atomic
  - ⇒ `shared` can suddenly end up with an odd value
Avoiding Inconsistencies

Always use Atomic Actions
- not always possible for certain operations
- hard to tell if an operation is atomic
  → depends on compiler and system implementation

Protect Critical Regions
- use *synchronisation constructs* before accessing shared resources
  → transforms operations into atomic actions
Example (turn-based mutual exclusion)

```c
int turn = 0;
int shared = 0;

void task1(void)
{
    while (turn != 0) // do nothing
    {
        // critical section
        shared++;
        shared++;
    }
    turn = 1;
}

extern int turn;
extern int shared;

void task2(void)
{
    while (turn != 1) // do nothing
    {
        // critical section
        shared += 2;
        // end critical section
    }
    turn = 0;
}
```
Analysis of Attempt #1

- **Guarantees Mutual Exclusion**
- **Drawbacks**
  - tasks are forced to strictly alternate their use of the shared resource
    - \( \Rightarrow \) pace is dictated by the slower process
  - if one Task fails even outside the critical region, the other Task is stuck forever
  - Waiting Task consumes 100% CPU time
    - \( \Rightarrow \) Busy Waiting
Example (flag-based mutual exclusion)

```
int flag[2] = {FALSE, FALSE};
int shared = 0;

void task1(void)
{
    while (flag[1])  // do nothing
        flag[0] = TRUE;
    // critical section
    shared++;
    shared++;
    flag[0] = FALSE;
}

extern int flag[2];
extern int shared;

void task2(void)
{
    while (flag[0])  // do nothing
        flag[1] = TRUE;
    // critical section
    shared += 2;
    // end critical section
    flag[1] = FALSE;
}
```
Analysis of Attempt #2

- Task failing outside Critical Section
  → no longer affects the other task!
- Mutual Exclusion not guaranteed:
  - Task 0 enters and exits `while()` because `flag[1]` is FALSE
  - Task 1 enters and exits `while()` because `flag[0]` is FALSE
  - both set their flags and enter critical section!
    ⇒ flags are set too late!
Attempt #3

Example (setting flags first)

```c
int flag[2] = {FALSE, FALSE};
int shared = 0;

void task1(void)
{
    flag[0] = TRUE;
    while (flag[1])
        ; // do nothing

    // critical section
    shared++;
    shared++;
    flag[0] = FALSE;
}

extern int flag[2];
extern int shared;

void task2(void)
{
    flag[1] = TRUE;
    while (flag[0])
        ; // do nothing

    // critical section
    shared += 2;
    // end critical section
    flag[1] = FALSE;
}
```

René Hexel

Task Synchronisation
Analysis of Attempt #3

- **Mutual Exclusion guaranteed**
  - only one Task enters critical section at a time
- **Deadlock** can occur:
  - both tasks set their flags to **TRUE**
  - both tasks enter their `while()` loops and wait indefinitely for the other task to clear its flag!
  - no task will ever be able to do anything useful again.
Example (backing off)

```c
int flag[2] = {FALSE, FALSE};
int shared = 0;

void task1(void)
{
    flag[0] = TRUE;
    while (flag[1]) {
        flag[0] = FALSE;
        // delay a bit
        flag[0] = TRUE;
    }
    // critical section
    shared++;
    shared++;
    flag[0] = FALSE;
}

void task2(void)
{
    flag[1] = TRUE;
    while (flag[0]) {
        flag[1] = FALSE;
        // delay a bit
        flag[1] = TRUE;
    }
    // critical section
    shared += 2;
    // end critical section
    flag[1] = FALSE;
}
```

René Hexel
Analysis of Attempt #4

- Close to a correct solution
  - mutual exclusion guaranteed, no Deadlock
- Livelock can occur:
  - both tasks set their flags to TRUE
  - both tasks check the their task’s flag (TRUE)
  - both tasks release their flag and start again
  → endless loop grabbing and releasing their flag, consuming 100% of (useless) CPU time
Peterson’s Algorithm

Example (backing off)

```c
int flag[2] = {FALSE, FALSE};
int turn = 0;

void task1(void)
{
    flag[0] = TRUE, turn = 1;
    while (flag[1] && turn==0)
        ; // do nothing

    // critical section
    shared++;
    shared++;

    flag[0] = FALSE;
}

extern int flag[2];
extern int turn;

void task2(void)
{
    flag[1] = TRUE, turn = 0;
    while (flag[0] && turn==1)
        ; // do nothing

    // critical section
    shared += 2;

    flag[1] = FALSE;
}
```
Peterson’s Algorithm (2)

- Correct solution
  - mutual Exclusion, no Dead-/Livelocks
- Not a generic solution
  - works only for two tasks
    → still uses Busy Waiting
- Solution: Hardware and/or OS-Support
  - atomic Test-And-Set (TAS) CPU instructions
  - blocking a task w/o consuming CPU time
Semaphores

- Simple Signalling Mechanism
  - synchronisation of multiple Tasks
- Shared Integer Variable
  - usually initialised to nonnegative value
  - `Wait()` operation: `P()`
    - block task while semaphore $\leq 0$, decrement value
  - `Signal()` operation: `V()`
    - increment value, unblock task(s) on waiting queue
Semaphore Algorithm

Semaphore Operations

```c
int semaphore = 1;

P()
{
    while (semaphore <= 0)
        BLOCK;
    semaphore--;
}
```

```c
extern int semaphore;

V()
{
    semaphore++;
    WAKEUP;
}
```

- **P()** and **V()** cannot be interrupted!
- **BLOCK** enqueues a Task on the waiting queue
- **WAKEUP** removes the first Task from the waiting queue
Semaphore Advantages

→ **Flexibility!**
  - **Multiple tasks**
    - more than two tasks can be synchronised
  - **If initialised to an** $n > 1$
    - $n$ tasks can enter critical region!
  - **If initialised to an** $n < 1$
    - $-n + 1 \lor ()$ operations are required before first task can enter critical region!
Semaphores in C

Create and initialise a Semaphore

- `sem_open()`
  
  → `sem_t *s = sem_open("mysemaphore", O_CREAT, 0600, 1);`

- `P()`
  
  - `sem_wait()`

- `V()`
  
  - `sem_post()`
Task Synchronisation

- **Semaphores**
  - means for protecting critical regions
  - flexible method, handling more than one task

- **NSLock**  
  - Objective-C class
  - simple binary semaphore (0 and 1 values only)
  - always initialised to 1
  - `lock`
    - `P()` operation (set semaphore to 0)
  - `unlock`
    - `V()` operation (set semaphore to 1)
    - needs to be called by the task that called `lock`
    - `lock` must have been called before `unlock`
The Producer/Consumer Problem

Consider the following scenario:

- Infinite Array (buffer)
- Producer: Adds to buffer at position out
- Consumer: Reads from position in

Let’s look at a simple implementation
Producer/Consumer Code

Example (Producer)

```c
for(;;) // loop forever
{
    produce item v;
    buffer[in++] = v;
}
```

Example (Consumer)

```c
for(;;) // loop forever
{
    while (out >= in) ; // wait for buffer data
    consume(buffer[out++]);
}
```
Binary Semaphore Attempt

- **Binary Semaphore**
  - Protect Critical Region

- **Integer** \( n \)
  - \( n = \text{in} - \text{out} \)
  - Keeps Track of available buffer positions

**Straightforward Solution?**

→ Let’s look at the algorithm
Attempt #1

BinarySemaphore s = 1, d = 0;
int n = 0;

Producer
for(;;) {
    P(s);
    append();
    n++;
    if (n == 1)
        V(d);
    V(s);
}

Consumer
P(d);
for(;;) {
    P(s);
    take();
    n--;
    V(s);
    if (n == 0)
        P(d);
}
Consider the following

- Consumer has consumed all items: $n = 0 \quad d = 0$
- Producer adds another item: $n = 1 \quad d = 1$
- Consumer checks if $n == 0$: false $\quad d = 1$
- Consumer consumes new item: $n = 0 \quad d = 1$
- Consumer checks if $n == 0$: true $\quad d = 1$
- Consumer: P(d) returns immediately: $n = 0 \quad d = 0$
- Consumer reads non-existent item: $n = -1$
**Problems**

- $V(d)$ of Producer is not matched by $P(d)$ of Consumer!
  - Testing $n$ and then waiting is not atomic
  - Moving the test into the critical section
    - Makes it atomic
    - But introduces the possibility of a Deadlock!

*Solution*

- Set auxiliary variable $m$ inside critical region
Attempt #2

BinarySemaphore s = 1, d = 0;
int n = 0;

Producer

```
for(;;) {
    P(s);
    append();
    n++;
    if (n == 1)
        V(d);
    V(s);
}
```

Consumer

```
P(d);
for(;;) {
    P(s);
    take();
    int m = --n;
    V(s);
    if (m == 0)
        P(d);
}
```
Analysis of Attempt #2

- **Correct Solution**
  - No deadlocks can occur
  - $m$ was modified within the critical section
    - Atomic Action
    - Producer will not modify $m$
    - Test for $m$ is safe

- **Not very Elegant Solution**
  - Easy to mess up, requires a lot of helper variables

- **Better: Use Counting Semaphores**
Using Counting Semaphores

Semaphore $s = 1$, $n = 0$;

**Producer**

```c
for (;;) {
    P(s);
    append();
    V(s);
    V(n);
}
```

**Consumer**

```c
for (;;) {
    P(n);
    P(s);
    take();
    V(s);
}
```
Counting Semaphores Analysis

- **Elegant Solution**
  - No extra counters required

- **No Deadlocks**
  - Principle: grab Semaphores in same order
  - Release in reverse order
  - “Protector” Semaphore is innermost Semaphore
  - Each \( P(\ ) \) must be matched by a \( V(\ ) \)
  - → but match can be within another Task!
### Counting Extensions

- Arrays are not infinite
  - Add another counting Semaphore
  - Initialise to *capacity* of Array

- Minimum fill level
  - Consumer must wait until reached
  - Initialise counting Semaphore to negative value
    - $-n$ is the minimum fill level
  - Beware traditional semaphore implementations!
    - Allow only non-negative values!
Semaphore \( s = 1, \ min_i = -1, \ maxi = 10 \);
The Reader/Writer Problem

- **Shared Data Area**
  - Writer(s) write to the area
  - Reader(s) read from the area, but don’t consume
- Any number of Readers may simultaneously read
- Only one Writer may write at a time
- While a Writer is writing, no Reader may read
Reader/Writer Attempt #1

Semaphore  r = 1, w = 1;
int nread = 0;

**Reader**

```c
for(;;) {
    P(r);
    if (++nread == 1)
        P(w);
    V(r);
    READ();
    P(r);
    if (--nread == 0)
        V(w);
    V(r);
}
```

**Writer**

```c
for(;;) {
    P(w);
    WRITE();
    V(w);
}
```
Analysis of Attempt #1

- **Mutual Exclusion**
  - Writer or first Reader grabs writing semaphore
  - Readers can access simultaneously
    - Only first Reader needs to wait on $w$
  - Readers have Priority
    - Writers will block until there are no readers
  - Starvation of Writers
Semaphore x, y, z, w, r;
int nread = 0, nwrite = 0;

### Reader
```c
for(;;) {
    P(z); P(r); P(x);
    if (++nread == 1)
        P(w);
    V(x); V(r); V(z);
    READ();
    P(x);
    if (--nread == 0)
        V(w);
    V(x);
}
```

### Writer
```c
for(;;) {
    P(y);
    if (++nwrite == 1)
        P(r);
    V(y);
    P(w);
    WRITE();
    V(w);
    P(y);
    if (--nwrite == 0)
        V(r);
    V(y);
}
```
Writer Priority Analysis

- Readers still block writers
  - $P(w)$
- Waiting Writer blocks *new* Readers
  - $P(r)$
  - Outer Semaphore: takes precedence over $P(w)$
- No Starvation
  - $\Rightarrow$ Writers take Precedence
Deadlock and Starvation

- **Deadlock**
  - Permanent (cyclic) blocking of a set of tasks competing for shared resources

- **Starvation**
  - A condition in which a task gets delayed indefinitely (or for a significant period of time) because other tasks are always given preference
Deadlock Conditions

1. Mutual Exclusion
   - Only one task may enter a critical section

2. Hold and Wait
   - A task holds allocated resources while awaiting assignment of other resources

3. No Preemption
   - No resource can be forcibly removed from a task

4. Circular Wait
   - Closed chain of tasks, such that each task holds at least one resource needed by the next task in the chain
A Deadlock Occurs . . .
  . . . if all four conditions are met at the same time
⇒ Strategies need to tackle at least one of these conditions
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
Deadlock Prevention

- Excludes Deadlock Possibility
- Mutual Exclusion
  - cannot be disallowed!
- Hold and Wait
  - task must request all resources at once
- No Preemption
  - forcefully take away resources
- Circular Wait
  - Define a *linear ordering* of resource types
Deadlock Avoidance

- **Task Initiation Denial**
  - Do not start a task if its resource requirements do not meet available resources (and thus may cause a deadlock)

- **Resource Allocation Denial**
  - Banker’s Algorithm

- **Allows more Concurrency than Deadlock Prevention**
  - Dynamic Avoidance vs. static Prevention
Deadlock Detection

- Check for Deadlocks
  - At each resource allocation
- Less Conservative
  - better Resource Utilisation
  - after-the-fact detection of deadlocks
- Requires Recovery Strategy
  - abort all or some deadlocked tasks
  - checkpointing
  - preempt resources until Deadlock goes away
Dining Philosophers
Each Philosopher needs two chopsticks to eat

Deadlock:
-Everybody picks up one chopstick and waits for the other

Solution: Deadlock Prevention
-Number the chopsticks (linear ordering)
-Pick up chopstick with lower number first