Preview – Interprocess Communication

- Race Condition
- Mutual Exclusion with Busy Waiting
  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
  - Peterson’s Solution
  - Test and Set Lock – help from hardware
- Sleep and Wakeup
- The Producer-Consumer Problem
- Race Condition in Producer-Consumer Problem
- Semaphores
- The producer-consumer problem using semaphores
- Mutexes
- Monitors
Interprocess Communication

- Critical Section (Critical Region) – The part of program where the shared memory is accessed.
- Three issues related to InterProcess Communication (IPC)
  1. How one process can pass information to another
  2. How to make sure two or more processes do not get into the critical region at the same time
  3. Proper sequencing when dependencies are present (e.g. A create outputs, B consume the outputs)
Race Condition

- A situation where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, is called race condition.
Race Condition

- Process A tries to send a job to spooler, Process A reads \( \text{in} = 7 \), process A times out and goes to “blocked” state before updating \( \text{in} = \text{in} + 1 \).
- CPU switches to Process B. Process B also tries to send a job to spooler. Process B reads \( \text{in} = 7 \), writes its job name in slot 7, updates \( i \) to 8 and then goes to “blocked” state.
- CPU switches to Process A. Process A starts from the place it left off. Process writes its job name in slot 7, update \( i \) to 8.

Process B will never receive any output from the printer!!
Race Condition

Four conditions must hold for there to be a deadlock (Coffman et al. 1971):

1. No two processes may be simultaneously inside their critical sections (mutual exclusion).
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.
Mutual Exclusion with Busy Waiting

Mutual Exclusion Solutions
- Busy Waiting
- Sleep and Wakeup

Mutual Exclusion with Busy Waiting
- Disabling Interrupts
- Lock Variables
- Strict Alternation
- Peterson’s Solution
- Test and Set Lock – help from hardware
Mutual Exclusion with Busy Waiting
(Disabling Interrupts)

Disabling Interrupt

- Once a process enters the critical section, it disables all interrupts.
- With interrupts turned off the CPU will not be switched to another process the process finishes its job in the critical section.
- Unattractive – unwise to give user process to the power to turn off interrupts. If the process never turned it on again, it might cause the end of system.
Mutual Exclusion with Busy Waiting
(Lock Variables)

- Software solution
- There is a single shared variable (lock), initially 0.
- If lock = 0, the process sets it to 1 and enters the critical section.
- If lock = 1, the process will wait until it becomes 0. -- busy waiting –
- Unfortunately, it has the race condition (similar to the spooling directory problem).
Mutual Exclusion with Busy Waiting
(Lock Variables)

\[
\text{repeat} \\
\quad \text{while } \text{lock} \neq 0 \text{ do} \\
\quad \quad ; \text{ (no-operation)} \\
\quad \text{lock} = 1 \\
\text{until false}
\]

1. Initially, \( \text{lock} = 0 \).
2. A process \( P_1 \) tries to enter its critical section. A process \( P_1 \) checks lock value (= 0).
3. Before updating \( \text{lock} = 1 \), Process \( P_2 \) tries to enter its critical section. \( P_2 \) checks lock value (still = 0). \( P_2 \) sets \( \text{lock} = 1 \) and goes to its critical section.
4. \( P_1 \) is rescheduled. \( P_1 \) continues. \( P_1 \) sets \( \text{lock} = 1 \) and goes to its critical section.
5. Now \( P_1 \) and \( P_2 \) are in their critical sections at the same time.

Violating Mutual Exclusion!! (condition 1)
Mutual Exclusion with Busy Waiting
(Strict Alternation)

- Take turns
- For processor $P_i$ and $P_j$
- Variable turn can be $i$ or $j$.
- If $\text{turn} = i$, process $P_i$ can enter its critical section.
- Once $P_i$ finishes its job in its critical section, $P_i$ sets $\text{turn} = j$, let process $P_j$ enter its critical section.
Mutual Exclusion with Busy Waiting
(Strict Alternation)

repeat
  while turn \neq i do
    ; (no-operation)
  Critical Section (C.S.)
  turn = j;
  Non-Critical Section
until false

Initially turn = 0

1. P_0 is in its C.S. while P_1 is in its non-critical section.
2. P_0 finishes C.S. and sets turn = 1, P_1 is still in its non-critical section.
3. P_0 finishes its non-critical section and wants to go to its C.S. again but turn = 1.
4. P_0 might need to wait forever to enter its C.S.

Violating Condition 3!!

No process running outside its critical region may block other processes
Mutual Exclusion with Busy Waiting
(Peterson’s Solution)

```c
#define false 0
#define true 1
#define n 2
int turn
int interested[n]

void enter_region(int process);  
  {  
    int other;
    other = 1 – process
    interested[process] = true
    turn = process;
    while (turn == process &&
      interested[other] == true)
      ; /*no operation */
  }

void leave_region(int process)
  {
    interested[process] = false;
  }

void main()
  {
    repeat
      enter_region (int i)
      Critical Section
      leave_region (int i)
      Remainder Section
    until false
  }
```
Mutual Exclusion with Busy Waiting
(Peterson’s Solution)

1. Initially, neither process is in its critical section
2. Process $P_0$ calls enter_region(0)
   i. Set interested[0] = true;
   ii. Set turn = 0
3. Goes to critical section
4. Process $P_1$ calls enter_region(1) to enter its critical section
   i. Set interested[1] = true;
   ii. Set turn = 1;
   - Since interested[0] = true, it keeps on
Mutual Exclusion with Busy Waiting
(Peterson’s Solution)

Proof for Peterson’s Solution

Consider the case both P₀ and P₁ call enter_region(0) and enter_region(1) almost simultaneously.

interest[0] = true and interest[1] = true at the same time

But turn can only be turn = 0 or turn = . Whichever done last is the one that counts!!

Case 1) turn = 0

Inside enter_region(0)
Mutual Exclusion with Busy Waiting
(Test and Set Lock – hardware solution)

- TSL instruction - help from the hardware
- Instruction test and set lock
  TSL RX, LOCK
  1. Read the content at the memory address of lock into register RX.
  2. Store a non-zero value at the memory address of lock

- The operations of reading the lock and storing into register are guaranteed to be indivisible.

- How to use Test and Set Lock instruction to solve the race condition problem?
  - When lock = 0, any process may set lock = 1 using TSL instruction and go to its critical section.
  - When the process finishes its critical section, set lock = 0 using the original move instruction.
Sleep and Wakeup

- Instead of busy waiting, it goes to sleeping state.
- Once a process finishes its critical section, it calls wakeup function which allows one of the sleeping processes to enter its critical section.
The Producer-Consumer Problem

Description
- Two processes share a common, fixed-sized buffer.
- Producer puts information into the buffer, and consumer takes it out.

Trouble arises
- When the producer wants to put a new item in the buffer, but it is already full.
- When the consumer tries to take a item from the buffer, but buffer is already empty.
The Producer-Consumer Problem

- When the producer wants to put a new item in the buffer, but it is already full.
  - Solution – producer goes to sleep, awakened by consumer when consumer has removed one or more items.

- When the consumer tries to take an item from the buffer, but buffer is already empty.
  - Solution – consumer goes to sleep, awakened by the producer when producer puts one or more items in the buffer.
The Producer-Consumer Problem

```c
#define N 100
int count = 0;
void producer()
{
    int item
    while (true)
    {
        item = produce_item();
        if (count == N)
            sleep();
        insert_item(item)
        count = count + 1;
        if (count == 1)
            wakeup(consumer);
    }
}

void consumer()
{
    int item;
    while(true)
    {
        if (count == 0)
            sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1)
            wakeup(producer);
        consume_item(item);
    }
}
```
Race condition in producer-consumer problems

1. Initially buffer is empty (count = 0)
2. The consumer reads count = 0. Since the consumer’s CPU time is over, scheduler assigns CPU time to producer.
3. Producer produces item and checks count, count = 0. Insert item to buffer. Increase count to count +1. If count =1, it wakes up consumer. Since the consumer is not sleeping yet, too bad, consumer misses the wakeup signal.
4. The consumer gets CPU time. Consumer reads count =0, consumer goes to sleep.
5. the producer keeps producing items and finally buffer becomes full. The producer goes to sleep.
A semaphore is an integer variable which could have value of
- 0: if no wakeups are saved
- i: if i wakeups are pending

A semaphore is accessed only through two standard atomic operations *down* (or P) and *up* (or V).
Concept of Semaphores

- Modification to the integer value of the semaphore in the `down` and `up` operations are executed *indivisibly*. It is all done as a single atomic action.

- It means that when a process is modifying the semaphore value, no other process can simultaneously modify the same semaphore value.
Semaphore Operation

void down (S)
{
    if S ≤ 0
    {
        1. add this process to the sleeping list
        2. blocked
    }
    S = S – 1;
}

void up (S)
{
    S = S + 1;
    if S = 1
    {
        1. choose one process P from the sleeping list
        2. wakeup(P) to finish its down operation
    }
}
Semaphore Implementation

A normal way for implementing a semaphore

- Implement semaphore operations *up* and *down* as system calls.
- Disables all interrupts while it is testing the semaphore, updating it, and putting the process to sleep.
Usage of semaphores

semaphore mutex = 1

repeat
down (mutex);

Critical Section

up (mutex);

Remainder Section

until false

void down (S)
{
    if (S <= 0)
    {
        1. add this process to the sleeping list
        2. blocked
    }
    S = S - 1;
}

void up (S)
{
    S = S + 1;
    if (S == 1)
    {
        1. choose one process P from the sleeping list
        2. wakeup(P) to finish its down operation
    }
}
Solving the Producer-Consumer Problem using Semaphores

```c
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer()
{
    int item;
    while (true)
    {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer()
{
    int item;
    while (true)
    {
        down(&full)
        down(&mutex)
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```
Careless usage of Semaphore causes deadlock

```c
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer ()
{
    int item;
    while (true)
    {
        item = produce_item();
        down (&mutex);
        down (&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer()
{
    int item;
    while (true)
    {
        down(&full)
        down(&mutex)
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```
Mutexes

- When the semaphore's ability to count is not needed, the simplified version of the semaphore, called mutex is used.
- It is good for managing mutual exclusion to some shared resources or pieces of code.
- A mutex is a variable that can be in one of the two states: unlocked (0), locked(1).
- A mutex concept is the same as binary semaphore which has value 0 or 1.
 Mutexes

mutexes mutex = 0
repeat
    mutex_lock (mutex);
    Critical Section
    mutex_unlock (mutex);
    Remainder Section
until false