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 in 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 creates outputs, B consumes 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 \( in = 7 \), process A times out and goes to “ready” state before updating \( in = in + 1 \).

- CPU switches to Process B. Process B also tries to send a job to spooler. Process B reads \( in = 7 \), writes its job name in slot 7, updates \( i \) to 8 and then goes to “ready” 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

A solution for the race condition should have following conditions:

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

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 the power to turn off interrupts. If the process never turned it on again, it might cause the end of the system.
Mutual Exclusion with Busy Waiting

(Lock Variable)

- **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 Variable)

repeat
    while lock ≠ 0 do
        ; (no-operation)
    lock = 1

Critical Section

lock = 0;

Remainder Section

until false

Violating Mutual Exclusion !! (condition 1)

1. Initially, lock = 0.
2. A process P₁ tries to enter its critical section. A process P₁ checks lock value (= 0).
3. Before updating lock = 1, Process P₂ tries to enter its critical section. P₂ checks lock value (still = 0). P₂ sets lock = 1 and goes to its critical section.
4. P₁ is rescheduled. P₁ continues. P₁ sets lock = 1 and goes to its critical section.
5. Now P₁ and P₂ are in the their critical sections at the same time.
Mutual Exclusion with Busy Waiting
(Strict Alternation)

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

repeat
  while turn ≠ i do
    ; (no-operation)
  Critical Section
  turn = j;
  Non-Critical Section
until false

Initially turn = 0
1. P₀ is in its C.S. while P₁ is in its non-critical section.
2. P₀ finishes C.S. and sets turn = 1, P₁ is still in its non-critical section.
3. P₀ finishes its non-critical section and wants to go to its C.S. again but turn = 1.
4. P₀ 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)
        Non-Critical Section
    until false
}
```
Mutual Exclusion with Busy Waiting
(Peterson’s Solution)

1. Initially, neither process is in the critical section
2. P₀ calls enter_region (0)
   1. Set interested[0] = true;
   2. Set turn = 0
3. go to critical section
4. P₁ calls enter_region(1) to get into its critical section
   1. set interested[1] = true;
   2. set turn = 1;
5. since interested[0] = true, it keeps looping for interested[0] = false
6. (finally) P₀ finishes its critical section and calls leave_region(0)
   1. set interested[0] = false
7. P₁ finds out interested[0] = false, P₁ goes to its critical section
Mutual Exclusion with Busy Waiting
(Peterson’s Solution)

Proof for Peterson’s Solution

- Consider the case that both P0 and P1 call enter_region(0) and enter_region(1) almost simultaneously.
- Two statements of “interested[0]= true” and “interested[1] = true” are done.
- Note that: For the two statements of “turn = 0”(in enter_region(0)) and “turn = 1” (in enter_region(1)), whichever done last is the one that counts!!

Case 1) “turn = 0” (done last)

Inside enter_region(0)
- Since turn =0 and interested[1] = true, P0 keeps looping in no-operation while-loop until P1 sets interested[1] = false (that is when P1 finishes its critical section).

Inside enter_region(1)
- Since turn = 0 and interested[0] = true, P1 goes to its critical section.

Case 2) “turn = 1” (done last)

Inside enter_region(0)
- Since turn =1 and interested[1] = true, P0 goes to its critical section

Inside enter_region(1)
- Since turn = 1 and interested[0] = true, P1 keeps looping in no-operation while-loop until P0 sets interested[0] = false (that is when P1 finishes its critical section).
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 content of lock and storing into it are guaranteed to be indivisible.

- How to use Test and Set Lock instruction for solving race condition?
  - 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 sleeping process to get into 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.

Troubles 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 into 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 check 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. Consumer gets CPU time. Consumer reads count =0. *Consumer goes to sleep!*
5. Producer keeps producing items and finally buffer becomes full. *Producer goes to sleep!*
A semaphore is an integer variable which could have value
- 0: no wakeups are saved
- + i: 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**.

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

void down (S) {
    If S \leq 0 {
        1. Add this process to the sleeping list
        2. block;
    } 
    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 down operation
    }
Semaphore Implementation

The normal way for implementing a semaphore

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

semaphore \texttt{mutex} = 1
repeat
  \texttt{down (mutex)};
  \texttt{Critical Section}
  \texttt{up (mutex)};
  \texttt{Remainder Section}
\texttt{until false}

\begin{verbatim}
void down (S)
{
  If S \leq 0
  {
    1. Add this process to the sleeping list
    2. block;
  }
  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 down operation
  }
}
\end{verbatim}
Solving the Producer-Consumer Problem using Semaphores

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.
- Good for managing mutual exclusion to some shared resources or pieces of code.
- Useful in thread packages that are implemented in user’s space.
- Mutex is a variable that can be in one of two state: unlocked (0), locked(1).
- 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