Section I
Section Real Time Systems. Processes

1.4 Inter-Processes Communication (part 1)

• Process interaction

  • Processes often need to interact with each other in order to complete more complex tasks

  • Competition for the use of limited resources such as processor(s), or memory

• Co-operation (data exchange)

  ⇒ need for some form of inter-process synchronisation and/or communication

• Inter-process competition

  • Resources are:

    - Shareable (may be used by several processes concurrently) e.g. CPU(s), some data protected against modification (read-only access), etc.

    - Non-shareable (may be used by a single process at a time) e.g. peripherals, some data subject to modification (read-write access), etc.
• **Problem example:**

- A memory location that contains a variable accessible to more than one process

- If one process tests the variable while another modifies its value, the result for the first process is unpredictable

• **Inter-process co-operation**

• **Problems:**

  - Processes run *asynchronously* with respect to each other

  - Speed of one process relative to another is *unpredictable*

  - In order to co-operate, there are *certain points* at which processes must synchronise

• **Synchronisation:**

  - **Shared memory:** critical sections, semaphores, locks, flags

  - **Message passing:** send and receive
• **Problem example: shared queue**

  - IN and OUT are two shared variables
  - seven processes access the shared queue:
    - A, B, C – producers (enqueue items)
    - W, X, Y, Z – consumers (dequeue items)

  ![Diagram of shared queue with processes A, B, C, W, X, Y, Z accessing IN and OUT variables]

  - Is there any problem?

  - **process A:**
    - reads IN variable (1)
    - enqueues a new item (3)
    - updates IN value (5)

  - **process B:**
    - reads IN variable (2)
    - enqueues a new item (4)
    - updates IN value (6)
• **General synchronisation problem**

• “Race Condition”
  
  - Two or more processes are reading or writing some shared data
  
  - The final result depends on when (precisely) each process runs

• **Solution:**

```
Mutual Exclusion

→

CODE    DATA

↓        ↓

"Critical Sections"    "Record locking"
```
- **Critical Section** = part of a code where the shared memory is accessed

- If no two or more processes are in a common critical section at the same time ever, the race condition could be avoided

- **Locking Data** = shared data when accessed by a process could be locked, preventing other processes from accessing it

- Data is "unlocked" when the process finishes processing the shared data

- Performed in general inside the critical section

- **Avoiding race condition is not sufficient**

- **Conditions:**

  1. No two processes may be simultaneously inside their critical sections

  2. No assumptions may be made about relative process speeds or number of CPUs

  3. No process running outside its critical section may block other processes

  4. No process should have to wait forever to enter its critical section
• **Critical Sections’ Implementation**

```c
enter_region();
    /* critical section */
leave_region();
```

• Before entering its critical section, the process calls `enter_region()`

  - Causes the process to wait only when another process is currently in its critical section until it exit from it

• After the critical section, the process calls `leave_region()`

  - Indicates that the process has finished with the shared data & allows another process to enter

• **Mutual exclusion** can be achieved by enclosing each critical section by `enter_region()` and `leave_region()`

• **Implementation** of `enter_region()` and `leave_region()`:

  - A) Disable Interrupts?    D) Peterson's Idea?
  - B) Lock Variables?       E) TSL Instruction?
  - C) Strict Alternation?
A) Disable Interrupts

- Each process disables all interrupts just after entering its critical region and re-enables them just before leaving it.

- It is a solution as only via interrupts process switching may be triggered.

```c
enter_region()
{ disable_interrupts;}
leave_region()
{ enable_interrupts;}
```

- It is unwise to give user processes the power to disable interrupts:
  - Suppose one process disables the interrupts and never re-enables them again
    ⇒ The whole system will never work again
  - If the computer has more than one CPU, disabling interrupts affects only the CPU that executed the disable instruction
    ⇒ Other CPUs will continue to run processes that can access the shared memory

- Very simple solution, but **NOT** recommended
B) Lock Variables

- Requires usage of a single shared boolean lock variable (“occupied”), initialised with FALSE

- if occupied is FALSE entry to critical section is allowed, whereas if occupied is TRUE entry to critical section is NOT allowed

```c
enter_region()
{
    while(occupied == TRUE) /*test*/
    ; /* waiting */
    occupied = TRUE; /* lock */
}

leave_region()
{
    occupied = FALSE; /* unlock */
}
```

- A process that wants to enter a critical section, checks first to see if the entry is allowed and if it is not, the process waits in a tight loop

- Continuously testing a variable waiting for some value to appear is denoted as “busy waiting”

⇒ Wastes CPU time
• **Advice:**
  
  - It could be used only when there is a reasonable expectation that the wait is short

• **Problem:**
  
  - In practice it does not work

• **Reason:**
  
  - The separation of the **test** and the **lock** of the shared **occupied** variable, *(enter_region() is NOT “atomic”)*

  - Two concurrent processes may each find that **occupied** is FALSE before either has the chance to set it to TRUE

  \[\Rightarrow\text{ both processes enter the critical section at the same time}\]

• **Conclusion:**
  
  - Simple solution, but **NOT recommended**
C) Strict Alternation

- Requires usage of a single shared integer variable ("turn"), initialised with 0

- Before entering in the critical section, each process checks whether it is its turn (\texttt{turn} == \texttt{process\_no}) and if it is, it enters, otherwise busy waits for its turn

\begin{verbatim}
int turn = 0; /* init */

enter\_region \(\text{(int process\_no)}\)
{
    while (turn != process\_no)
        ; /* busy waiting */
}

leave\_region \(\text{(int process\_no)}\)
{
    turn = (++process\_no \% PROC\_COUNT);
}
\end{verbatim}

- PROC\_COUNT is the number of processes

- Disadvantages:
  - Processes may enter critical sections only in fixed order of process number
  - Uses busy waiting

- Conclusion:
  - Works, but it is very restrictive & wastes CPU
D) Peterson's Solution

- **Combines** the ideas of **lock variables** and taking turns

- **Does not** require strict alternation

```c
#define FALSE 0
#define TRUE 1
#define N 2 /* no processes */

int turn = 0; /* init turn */
int interested[N]; /* init FALSE */

enter_region(int process_no)
{
    /* the other process_no */
    int other_no;

    other_no = 1 - process_no;

    /* indicate the interest entering*/
    interested[process_no] = TRUE;

    turn = process_no; /* set turn */

    while (turn == process_no &&
           interested[other_no] == TRUE)
    ; /* busy waiting*/
}
```
leave_region(int process_no) 
{
    /* indicate leaving critical region */
    interested[process_no] = FALSE;
}

- **When both processes call enter_region at the same time,** whichever stores **turn** first is the process that enters the critical section first

- **Does not** require strict alternation

- **Disadvantages:**
  - Uses busy waiting
  - Difficult to generalise to N > 2 (complexity!)

- **Advantages:**
  - Works fine for two processes
  - Does not impose strict alternation

- **Conclusion:**
  - **Recommended** for two concurrent processes only
E) "TSL" (Test and Set Lock) Instruction

- Requires usage of a shared integer variable ("occupied") that co-ordinates access

  ```c
  int occupied;
  ```

- Uses a function ("tsl") that:
  - receives as parameter a variable transmitted "by-address"
  - stores a non-zero value in the variable
  - returns the old value of the variable

  ```c
  int tsl(int *variable);
  ```

  - reading the variable and storing values into it are guaranteed to be indivisible ("atomic")

  - no other processor can access the shared variable until the "tsl" instruction has finished
```c
enter_region()
{
    while (tsl(&occupied))
        ; /* busy waiting */
}

leave_region()
{
    occupied = FALSE;
}
```

- **Disadvantages:**
  - Uses busy waiting
  - Requires hardware support for “tsl” instruction implementation
  - Not easy to generalise

- **Advantage:**
  - Works fine for any number of processes

- **Conclusion:**
  - Highly recommended if “tsl” instruction is available
• Possible software `tsl()` implementation

```c
int sw_tsl(int *variable) {
    disable_interrupts();
    old_value = *variable;
    *variable = TRUE;
    enable_interrupts();
    return(old_value);
}
```

- Difficult to restrict user process access by using `disable_interrupts()`

• Possible implementation of `tsl()` as an Operating System (OS) Call

- Place the code for `tsl()` in OS

- OS can use H/W or S/W, as appropriate

- Any user process may use an OS call to access TSL service

- Protection hardware allows OS to disable the interrupts, but not also user processes!

- It is better, but `tsl()` still uses busy waiting

- Especially for single CPU systems this may have a huge negative impact
• **Priority Inversion Problem:**

- Could be seen as a side effect of busy waiting
- Situation: two processes - H ("high" priority) and L ("low" priority)
- Scheduling: H always runs when it is ready
- Suppose: L is already in its critical region, and H becomes ready to run (input some data)
- Effect:
  - H is run by scheduler, and reaches beginning of its critical section
  - H must busy-wait as L is in its critical section
  - Because of scheduling rules, L will never be run again to complete its critical section, so H waits forever!

• **Recommendation:**

- Prefer inter-process communication system calls that **block the process, instead of wasting CPU time**, when they are not allowed to enter their critical sections
  
  ```
  sleep() and wakeup()
  semaphores
  message passing
  ```