Interprocess Communication

Issues of concern:
1. How one process can pass information to another
2. Make sure that processes do not interfere with each other when engaging in critical activities.
3. Proper sequencing when dependencies are present

Race condition
A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

To avoid race conditions:
• Prohibit more than one process from reading and writing the shared data at the same time.

• busy waiting or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as waiting for keyboard input or waiting for a lock to become available

Critical Region/Section
A section of code within a process that requires access to shared resources and which may not be executed while another process is in a corresponding section of code.

Deadlock
A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

Starvation
A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Mutual Exclusion
The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
For example, only one process at a time is allowed to send command to the printer

Requirements for Mutual Exclusion
Four conditions to provide mutual exclusion
1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process (deadlock)
4. No process must wait forever to enter its critical region (starvation)

Mutual Exclusion:

Hardware Support

Interrupt Disabling
• A process runs until it invokes an OS service or until it is interrupted. Disabling interrupts guarantees mutual exclusion

• Disadvantages
– Unwise to allow user processes to turn off interrupts. What if one of them did it and never turned them on again?
– Multiprocessing: disabling interrupts on one processor will not guarantee mutual exclusion
• Often a useful technique within the OS itself but not appropriate as a general mutual exclusion mechanism for user processes.

• Special Machine Instructions
– Performed in a single instruction cycle
– Access to the memory location is blocked for any other instructions
Atomic action is one which cannot be interrupted
– either because it is performed by a single CPU instruction that locks the memory bus, or
– because it blocks the interrupt mechanism while the operation is running.
Atomic actions are important in issues of process synchronization and mutual exclusion.

Mutual Exclusion Machine Instructions

• Advantages
– Applicable to any number of processes on either a single processor or multiple processors sharing main memory
– It is simple and therefore easy to verify
– It can be used to support multiple critical sections
• Disadvantages
– Busy-waiting consumes processor time
– Starvation is possible when a process leaves a critical section and more than one process is waiting.
– Deadlock
• If a low priority process has the critical region and a higher priority process needs it, the higher priority process will obtain the processor to wait for the critical region

Peterson’s Solution
Two process solution
• Assume that the LOAD and STORE instructions are atomic (cannot be interrupted)
• The two processes share two variables:
– int turn; – boolean interested[2]
• The variable turn indicates whose turn it is to enter the critical section.

• The interested array is used to indicate if a process is ready to enter the critical section.
– interested[i] = true implies process Pi is ready!

Semaphores
• Synchronization tool that does not require busy waiting
• Less complicated
• First defined by Dijkstra in late 60s
• Main synchronisation principle used in original Unix
• Semaphore S – integer variable used for signaling
– If a process is waiting for a signal, it is suspended until that signal is sent
• Two standard atomic operations modify S: wait() and signal()
– Originally called P() and V()
Semaphore is a variable that has an integer value
– May be initialised to a nonnegative number
– Wait operation decrements the semaphore value
– Signal operation increments semaphore value
• Counting semaphore: integer value can range over an unrestricted domain
• Binary semaphore: integer value can range only between 0 and 1; can be simpler to implement
– Also known as mutex locks
• Can implement a counting semaphore S as a binary semaphore

Semaphore Implementation with no Busy waiting
• With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
– value (of type integer)
– pointer to next record in the list
• Two operations:
– block – place the process invoking the operation on the appropriate waiting queue.
– wakeup – remove one of processes in the waiting queue and place it in the ready queue.

A big step but…
• They are essentially shared global variables.
• Access to semaphores can come from anywhere in a program
• They serve two purposes, mutual exclusion and scheduling constraints.
• There is no control or guarantee of proper usage
• To ensure that wait() and signal() are executed atomically, interrupts need to be disabled on every processor

General rule: use a separate semaphore for each constraint

Readers-Writers Problem
Dining Philosophers Problem

Problems with Semaphores
• Using semaphores incorrectly can result in timing errors that are difficult to detect
• Correct use of semaphore operations: .
– all processes share a semaphore variable mutex,
– mutex is initialised to 1.
– each process must execute wait(mutex) before entering the critical section and singal(mutex) afterwards.
• Incorrect use leads to:
– signal (mutex) …. wait (mutex) mutual exclusion violation
– wait (mutex) … wait (mutex) deadlock
– omitting of wait (mutex) or signal (mutex) (or both) mutual exclusion violation or deadlock

Monitors

• A high-level abstraction that provides a convenient and effective mechanism for process synchronisation
• Only one process may be active within the monitor at a time