Chapter 5: Process Synchronization

Cooperating Process:

- Can affect or be affected by other processes executing in the system
- Can either share logical address space (code and data) or data only -> thru file and messages
- Concurrent access can result in data inconsistency

Race Condition: where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place -> consumer and producer algorithms are an example.

Critical Section Problem

Protocol that processes can use to cooperate. N processes with the following structure:

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

No other process is allowed to execute. Permission is required to enter.

Process may be changing common variables, updating a table, writing a file

Requirements:

- *Mutual exclusion: when process is executing in its critical section, no other can do it.*
- *Progress: only processes not executing in their reminder sections can participate in deciding which will enters its critical section.*
- *Bounded waiting: limit on the number of times other process are allowed to enter critical sections.*

Kernel data structures prone to possible race conditions are the ones

- ✓ Maintaining memory allocation
- ✓ Maintaining process lists
- ✓ Interrupt handling

Approaches to handle critical sections:

- ✓ Preemptive kernels: Allows a process to be preempted while it is running in kernel mode. May be more responsive / More suitable for real time programming
- ✓ Non preemptive kernels: a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU -> *free from race conditions* on kernel structures (only one process is active at a time)

Paterson's solution

Restricted to 2 processes that alternate execution between their critical sections and reminder sections.

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Two processes share:

```
int turn;
boolean flag[2];
```

Whose turn is.

If a process is ready to enter in its critical section

They are not guarantee to work on modern computer architectures

Synchronization Hardware

Solutions based on the premise of locking -> protecting critical regions thru the use of locks.

One solution could be prevent interrupts from occurred while a shared variables was being modified) - > not feasible in multiprocessor environments, time consuming/ system efficiency decreases. Effect in the system clock if it is updated by interrupts.

Hardware instruction 1. Test and set

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}
```

Executed atomically. If 2 are executed simultaneously they will be executed sequentially in some arbitrary order.

```
do {
  while (test_and_set(&lock))
    ; /* do nothing */

    /* critical section */

  lock = false;

    /* remainder section */
} while (true);
```

Boolean variable to implement mutual exclusion

Hardware instruction 2.Compare and swap

```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;

  if (*value == expected)
    *value = new_value;

  return temp;
}
```

The operand value is set to new value only if the expression (*value == exected) is true. Regardless, compare and swap() always returns the original value of the variable value.

```
do {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */

    /* critical section */

  lock = 0;

    /* remainder section */
} while (true);
```

They don't satisfy the bounded-waiting.

Algorithm with test and set that satisfies all requirements

```
do {
  waiting[i] = true;
  key = true;
  while (waiting[i] && key)
    key = test_and_set(&lock);
  waiting[i] = false;

    /* critical section */

  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

  if (j == i)
    lock = false;
  else
    waiting[j] = false;

    /* remainder section */
} while (true);
```

Common data Structures:

```
boolean waiting[n];
boolean lock;
```

Initialized to false.

Mutual exclusion –DONE- waiting[i] == false or key == false

Progress requirement -DONE- either sets lock to false or sets waiting[j] to false

Bounded waiting requiremt –DONE- when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n − 1, 0, ..., i − 1). It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section.

Synchronization tool: Mutex Locks (spinlock)

```
acquire() {
    while (!available)
      ; /* busy wait */
    available = false;;
}
```

The acquire()function acquires the lock, and the release() function releases the lock. Available indicates if the lock is available.

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

Release function just set the available variable to TRUE.

Disadvantages: Busy waiting- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().

Advantage: no context switch is required when a process must wait on a lock, and a context switch may take considerable time.

Synchronization tool: *Semaphores*

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

```
wait(S) {
    while (S <= 0)
      ; // busy wait
    S--;
}
```

Modifications to S must be executed indivisibly and without interruption

```
signal(S) {
    S++;
}
```

- Counting Semaphore
  - Range over unrestricted domain
  - Used to control access to a given resource consisting of a finite number of instances
  - Initialized to the number of resources available
  - Wait() -> to use a resource
  - Signal() -> release a resource
  - When S=0 all resources are being used (block until count becomes greater)
- Binary Semaphore
  - range between 0 and 1 -> similar to mutex behavior

These implementations suffer from busy waiting. To overcome this situation:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

- when a process must wait is added to list of processes
- Signal() removes from list of processes

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- When semaphore value is not positive it should block itself (places a process in a waiting queue- **state**: waiting)
- CPU scheduler should select another process to execute
- Wakeup() restarts a process that is blocked when other executes signal() -> **state:** ready / place in ready queue
- Block() and wakeup() provided as System calls

- S may be negative
- FIFO queue to remove or add to porcess list ensures bounded waiting
- Operations should be exceuted atomically -not 2 porcesses can execute wait() and signal()-
- The implementation doesn't remove busy waiting entirely

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Deadlock: situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

Starvation: a situation in which processes wait indefinitely within the semaphore.

Priority inversion: when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process -> **Priority inheritance protocol** is used to solve it (all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question)

Classic Problems of synchronization

- Bounded Buffer Problem
- The readers writers problem
- The dining-philosophers problem

Monitors – to deal with errors caused by incorrect use of semaphores

```
monitor monitor name
{
  /* shared variable declarations */

  function P1 ( . . . ) {
     . . .
  }

  function P2 ( . . . ) {
     . . .
  }
        .
        .
        .
  function Pn ( . . . ) {
     . . .
  }

  initialization_code ( . . . ) {
     . . .
  }
}
```

Abstract Data Type (ADT): encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.

- Monitor Type is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor.
- Cannot be used directly by the various processes.
- Local variables access by local functions
- Only one process at a time is active within the monitor
- Not powerful enough -> **solution**: condition construct
  - Variables of type condition : condition x,y ;
  - Can invoke wait() and signal() -> x.wait()
  - The x.signal() operation resumes exactly one suspended process
- To implement a monitor using semaphores a semaphore mutex is provided. A process executes wait(mutex) before entering the monitor and signal(mutex) after leaving the monitor. For conditions we introduce a semaphore x_sem and x_count (initialized to 0).

```
wait(mutex);
   ...
   body of F
   ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

x.wait()

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

x.signal()

```
monitor DiningPhilosophers
{
  enum {THINKING, HUNGRY, EATING} state[5];
  condition self[5];

  void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
      self[i].wait();
  }

  void putdown(int i) {
    state[i] = THINKING;
    test((i + 4) % 5);
    test((i + 1) % 5);
  }

  void test(int i) {
    if ((state[(i + 4) % 5] != EATING) &&
     (state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING)) {
       state[i] = EATING;
       self[i].signal();
    }
  }

  initialization_code() {
    for (int i = 0; i < 5; i++)
      state[i] = THINKING;
   }
}
```

Monitor solution to the dining-philosopher problem

Synchronization Examples

- Windows
  - Spinlocks to global resources access
  - A thread will never be preempted while holding a spinlock
  - Outside the kernel : dispatcher objects -> mutex locks, semaphores, events (condition variables) and timers (notify that a specified amount of time has expired)
  - Dispatcher objects states: **signaled (**available, thread will not block when acquiring the object **)** state or **non-signaled** (not available, will block) state
  - A critical-section object is a user-mode mutex that can often be acquired and released without kernel intervention.
- Linux
  - Atomic integer technique – atomic_t- all math operations using atomic integers are performed without interruption
  - Mutex locks for protecting critical sections within the kernel
  - Also provides spinlocks and semaphores for locking in the kernel
  - Approach to disable and enable kernel preemption: system calls—preempt disable() and preempt enable() and preempt count, to indicate the number of locks being held by the task.
- Solaris
  - Adaptive mutex locks, condition variables, semaphores, reader–writer locks, and turnstiles to control access to critical sections.
  - Adaptive mutex locks: On a multiprocessor system, an adaptive mutex starts as a standard semaphore implemented as a spinlock.

Alternative Approaches

- **Transactional memory.**

- Memory transaction: is a sequence of memory read–write operations that are atomic. If operations completed -> transaction is committed. Otherwise-> operations aborted and rolled back.
- Can be implemented in software or hardware (STM or HTM)

- **OpenMP**
  - includes a set of compiler directives and an API
  - Any code following the compiler directive **#pragma omp parallel** is identified as a parallel region and is performed by a number of threads equal to the number of processing cores in the system.
  - **#pragma omp critical**, which specifies the code region following the directive as a critical section in which only one thread may be active at a time.

- **Functional programming languages**
  - Functional languages do not maintain state. Because functional languages disallow mutable state, they need not be concerned with issues such as race conditions and deadlocks.
  - Erlang and scala are examples