Lecture 5 (IPC)

InterProcess Communication (or IPC) should be carried out in a well-structured way not using interrupts.

3 issues:

- how one process can pass information to another
- two or more processes don't get in each other's way of accessing same resource
- process sequencing and dependencies (if A ⇒ B, then B should wait A)

2 last applies also to threads

A **race condition** is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

Race condition - situation where two or more processes are reading & writing some shared data & final result depends on who runs precisely when

Avoiding race condition:

- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Semaphores
- Mutexes
- Monitors
- Message Passing
- Barriers
- Avoiding Locks: Read-Copy-Update

Mutual exclusion - way to prohibit more than one process from reading/writing shared data at the same time

Critical region (critical section) - part of the program where shared memory is accessed. If there is no any two processes in critical region ⇒ race condition avoided

Mutual exclusion use critical regions.

Requirements to critical regions:

1. no 2 process in their critical regions simultaneously
2. no assumption may be made about speeds or number of CPUs
3. no process running outside critical region may block any process
4. no process should wait forever to enter its critical region

Possible implementation of critical region:

1. disabling interrupts
2. lock variables
3. strict alternation
4. Peterson's solution
5. TSL instruction

Mutual Exclusion with Busy Waiting:

- Disabling interrupts (each process disable all interrupts after entering the critical region and re-enable before leaving). Correctly applicable only on 1 CPU - not appropriate as general solution
- Lock variables (shared variable - lock, lock==0 → set lock =1 and enter critical region, lock ==1 →wait until lock = 0) - can happen some fatal errors
- Strict alternation (integer variable - turn = 0; process 0 finds it 0 →enter critical region, process 1 find it 0 and sits in a tight loop waiting for its turn). Spin lock - lock that uses busy waiting) **VIOLATES CONDITION 3: NO PROCESS RUNNING OUTSIDE ITS CRITICAL REGION MAY BLOCK OTHER PROCESSES.**

```
while(TRUE){
  while (turn != 0)
```

```
while(TRUE){
   while(turn !=1)
```

```
  critical_region();                                    critical_region();
  turn = 1;                                             turn = 0;
  noncritical_region();                                 noncritical_region();
}                                                     }
//process 0                                           //process 1
```

- **Peterson's solution** (Before entering its critical region, each process calls **enter_region()** with its own process number, 0 or 1, as parameter. This call will cause it to wait until it is safe to enter. After it has finished with the shared variables, the process calls **leave_region()** to indicate that it is done)

```
int turn;                //which process
int interested[2]        //inside critical region or not

void enter_region(int process){                 void leave_region(int process){
  int other;                                      intersted[process] = 0;
  other = 1 - process;                          }
  interested[process] = 1;
  turn = process;
  while(turn == process && interested[other]==1);
}
```

- TSL(Test and set lock) instruction (The first instruction copies the old value of lock to
  the register and then sets lock to 1. The old value is compared with 0: If it is nonzero, the lock was already set by another program. Sooner or later it will become 0 and the subroutine returns, with the lock set. To clear the lock the program just stores a 0 in lock). **Hardware-assisted approach; works with multiple CPUs.**

```
enter_region:                           leave_region:
  TSL REGISTER,LOCK                        MOVE LOCK, #0
  CMP REGISTER, #0                         RET
  JNE enter_register
  RET
```

Continuously testing a variable until some value appears is called **busy waiting**. A lock that uses busy waiting is called a **spin lock.**

Peterson & TSL both have **defect of requiring busy waiting.**

Priority inversion problem - scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively inverting the relative priorities of the two tasks.

Simplest interprocess communication primitives that block instead of wasting CPU time are system calls **sleep** and **wakeup**. **Sleep** - syscall causes the caller to be suspended (blocked) until another process wakes it up. **Wakeup** - syscall accepts the process to be awakened as a parameter and wakes this process up (supposedly, changes its state to Ready)

The producer-consumer problem (bounded-buffer problem):

- Two processes share a common, fixed-size buffer
- One of them, the producer, puts information into the buffer
- The other one, the consumer, takes it out
- When the producer wants to put a new item in the buffer, but it is already full, it goes to sleep
- Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep too

```
N = 100;   //size of shared buffer
void producer(void){                              void consumer(void){
  int item;                                         int item;
  while(TRUE){                                      while(TRUE){
    item = produce_item();                             if (count == 0)sleep();
    if (count==N)sleep();                              item = remove_item();
    insert_item(item);                                 count--;
    count++;                                           if (count==N-1)wakeup(producer);
    if (count==1)wakeup(consumer);                     consume_item(item);
  }                                                 }
}                                                 }
//producer-consumer problem with fatal race condition
```

Problem: wakeup sent to a process that is not (yet) sleeping is lost. Fix - wakeup waiting bit. Wakeup sent - bit set.

Semaphore: - an integer variable to count the number of wake-ups saved for future use. Semaphore has a positive value if one or more wakeups were pending, 0 otherwise. Two operations: down and up (generalizations of sleep and wakeup,

respectively). The down operation checks if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues.

```
semaphore mutex = 1;      //control access to critical region
semaphore empty = N;      //counts empty buffer slots
semaphore full = 0;       //counts full buffer slots
void producer(void){                          void consumer(void){
  int item;                                     int item;
  while(TRUE){                                  while(TRUE){
    item = produce_item();                         down(&full);
    down(&empty);                                  down(&mutex);
    down(&mutex)                                   item = remove_item(item);
    insert_tem(item);                              up(&mutex);
    up(&mutex);                                    up(&empty);
    up(&full);                                     consume_item(item);
  }                                             }
}                                             }
//producer-consumer problem's solution with semaphores
```

Binary semaphore - initialized to 1 & used by 2+ processes to ensure that only one can enter its critical region.

Semaphores & synchronization: full & empty semaphores are needed to guarantee that certain event sequences do or don't occur.

A **mutex** is a shared variable that can be in one of two states: unlocked or locked.

Two **procedures** are used with mutexes:

- When a thread (or process) needs access to a critical region, it calls **mutex_lock**. If the mutex is currently unlocked, the call succeeds and the calling thread is free to enter the critical region
- If the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls **mutex_unlock**. If multiple threads are blocked on the mutex, one of them is chosen at random.

Futex - fast user space mutex - avoids dropping into the kernel unless it has to.

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Conditional variables allow to block due to some conditions not met.

Monitor is a higher-level synchronization primitive – it is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.

Only **one process** can be active in a monitor at any moment.

Message passing is the method of IPC that uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs.

Former call sends message to dest →latter receives a message from source. If no message ⇒receiver block until one arrives or return error.

The main issue: **messages can be lost by the network**. The receiver might not be able to distinguish a new message from the retransmission of an old one

---

Barrier: processes processed by phases. When a process reaches barrier, it's blocked until all have reached the barrier.

Read-copy-update(RCU):

RCU is synchronization mechanism that avoids the use of lock primitives while multiple threads concurrently read and update elements that are linked through pointers and that belong to shared data structures.

Specifically, readers access the data structure in what is known as a **read-side critical section** which may contain any code, as long as it does not block or sleep.

Grace period - time period when each thread to be outside read-side critical region at least once.