Communicating Processes

- Many applications require processes to communicate and synchronize with each other
- Main problem: operations of different processes are interleaved in an unpredictable manner
- Same issues in multiple contexts
  - Multiple threads of same process accessing shared data
  - Kernel processes accessing shared data structures
  - User processes communicating via shared files
  - User processes communicating via shared objects in kernel space
  - High-level programming languages supporting parallelism
  - Database transactions

Main Concerns

- How one process can pass information to another
- Making sure two or more processes do not get into each other’s way when accessing shared information
- Proper sequencing when dependencies are present

Example: Shared variable problem

- Two processes are each reading characters typed at their respective terminals
- Want to keep a running count of total number of characters typed on both terminals
- A shared variable V is introduced; each time a character is typed, a process uses the code:
  \[ V = V + 1; \]
  to update the count.
- During testing it is observed that the count recorded in V is less than the actual number of characters typed. What happened?

Analysis of the problem

The programmer failed to realize that the assignment was not executed as a single indivisible action, but rather as an arbitrary shuffle of following sequences of instructions:

P1. MOVE V, r0  Q1. MOVE V, r1
P2. INCR r0      Q2. INCR r1
P3. MOVE r0, V   Q3. MOVE r1, V

The interleaving P1, Q1, P2, Q2, P3, Q3 increments V only by 1

Sample Question

```c
void interleave() {
    pthread_t th0, th1;
    int count=0;
    pthread_create(&th0,0,test,0);
    pthread_create(&th1,0,test,0);
    pthread_join(th0,0);
    pthread_join(th1,0);
    printf("%d\n", count);
}

void test() {
    for(int j=0;j<MAX;j++) count=count+1;
}
```

What's minimum/ maximum value output?
Push and Pop example

```
struct stacknode {
    int data;
    struct stacknode *nextptr;
};
typedef struct stacknode STACKNODE;

void push (STACKNODE **topptr, int info) {
    STACKNODE *newptr = malloc(sizeof(STACKNODE));
    newptr->data = info; /* Push 1 */
    newptr->nextptr = *topptr; /* Push 2 */
    *topptr = newptr;          /* Push 3 */
}
```

```
int pop (STACKNODE **topptr) {
    STACKNODE *tempptr = *topptr;            /* Pop 1 */
    int popvalue = (*topptr)->data;   /* Pop 2 */
    *topptr = (*topptr)->nextptr; /* Pop 3 */
    free(tempptr);
    return popvalue;
}
```

Question: Is it possible to find an interleaved execution of Push 1, Push 2, ..., Pop 3 such that the resulting data structure becomes inconsistent?

Issues in Concurrent Programming

- Operations on shared data structures typically correspond to a sequence of instructions
- When two processes/threads are executing concurrently, the result depends on the precise interleaving of the two instruction streams (this is called race condition)
- Race conditions could cause bugs which are hard to reproduce
- Besides race condition, the second issue is synchronization (one process is waiting for the results computed by another)
  – Can we avoid busy waiting?

Overview of Solutions

<table>
<thead>
<tr>
<th>High-level Synchronization Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitors (Hoare, Brinch-Hansen)</td>
</tr>
<tr>
<td>Synchronized method in Java</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS-level support (mutual exclusion and synchronization)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special variables: Semaphores, Mutexes</td>
</tr>
<tr>
<td>Message passing primitives (send and receive)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low-level (for mutual exclusion)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt disabling</td>
</tr>
<tr>
<td>Using read/write instructions</td>
</tr>
<tr>
<td>Using powerful instructions</td>
</tr>
</tbody>
</table>

Requirements for solutions to Mutual Exclusion Problem

1. **Safety**: No two processes should be simultaneously in their critical regions
2. **Generality**: No assumptions should be made about speeds or numbers of CPUs (i.e., it should work in the worst case scenario)
3. **Absence of deadlocks**: Should not reach a state where each process is waiting for the other, and nobody gets to enter
4. **Bounded liveness (or fairness)**: If a process wants to enter a critical section then it should eventually get a chance

Mutual Exclusion

- Identify a block of instructions involving shared memory access called **critical region/section** (e.g., the entire assignment V=V+1; in our first example)

- Ensure that processes execute respective critical sections in a mutually exclusive manner, i.e. only one process is in its critical region at a time.
Low-level solution: Disable interrupts

<table>
<thead>
<tr>
<th>process A</th>
<th>process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable interrupts</td>
<td>disable interrupts</td>
</tr>
<tr>
<td>CS</td>
<td>CS</td>
</tr>
<tr>
<td>enable interrupts</td>
<td>enable interrupts</td>
</tr>
</tbody>
</table>

- Prevents context-switch during execution of CS
- This is sometimes necessary (to prevent further interrupts during interrupt handling)
- Not a good solution for user programs (too powerful and not flexible)

Shared Variable Solutions

General Skeleton

Two processes with shared skeleton
Assumption: Reads and Writes are atomic
Each process P0 and P1 executes

```c
/* Initialization */
while (TRUE) {
    /* entry code */
    CS() /* critical section */
    /* exit code */
    Non_CS() /* non-critical section */
}
```

No assumption about how often the critical section is accessed

1st Attempt for Mutual Exclusion

Shared variable: turn :{0,1}
turn==i means process Pi is allowed to enter
Initial value of turn doesn’t matter
Solution for process P0: (P1 is symmetric)
while (TRUE) {
    while (turn != 0); /* busy waiting */
    CS();
    turn = 1; /* let the other enter */
    Non_CS();
}

Ensures mutual exclusion, but requires strict alternation
A process cannot enter its CS twice in succession
even if the other process does not need to enter CS

2nd Attempt

Shared variable: interested[i] : boolean, init FALSE
Solution for process P0: (P1 is symmetric)
while (TRUE) {
    interested[0] = TRUE; /* declare entry first */
    while (interested[1]); /* wait if P1 is trying */
    CS();
    interested[0] = FALSE; /* unblock P1 */
    Non_CS();
}

Mutual Exclusion is violated:
P0 tests interested[1] and finds it False
P1 tests interested[0] and finds it False
Both proceed, set their flags to True and enter CS

3rd Attempt

Shared variable: interested[i] : boolean, init FALSE
Solution for process P0: (P1 is symmetric)
while (TRUE) {
    interested[0] = TRUE; /* declare entry first */
    while (interested[1]); /* wait if P1 is trying */
    CS();
    interested[0] = FALSE; /* unblock P1 */
    Non_CS();
}

Leads to deadlock:
P0 sets interested[0] to TRUE
P1 sets interested[1] to TRUE
Both enter their while loops and keep waiting

Peterson’s Solution

Shared variables: interested[i] : boolean; turn :{0,1}
Solution for process P0: (P1 is symmetric)
interested[0] = FALSE;
while (TRUE) {
    if (interested[1]) /* declare interest */
    turn = 0; /* take care of race condition */
    repeat if /* busy wait */
    (interested[1] == TRUE && turn == 0);
    CS();
    interested[0] = FALSE; /* unblock P1 */
    Non_CS();
}

P1 is contending, and it got there first:
turn = 1 executed before turn = 0
Proof of Mutual Exclusion

- To prove: P0 and P1 can never be simultaneously in CS
- Observation: Once P0 sets interested[0], it stays true until P0 leaves the critical section (same for P1)
- Proof by contradiction. Suppose at time t both P0 and P1 are in CS
- Let t0/t1 be the times of the most recent executions of the assignments turn = 0 / turn = 1 by P0 / P1, resp.
- Suppose t0 < t1
- During the period t0 to t, interested[0] equals TRUE
- During the period from t1 to t, turn equals to 1
- Hence, during the period t1 to t, P1 is blocked from entering its CS; a contradiction.

Also satisfies bounded liveness (why?)

Critical Regions

Peterson without Separate Code

```c
int turn, interested[2];

void enter_cs(int pid){
    int other = 1-pid;  /* pid is either 0 or 1 */
    interested[pid] = TRUE;
    turn = pid;
    while(turn==pid && interested[other]==TRUE);
}

void leave_cs(int pid) {
    interested[pid] = FALSE;
}
```

Using mutual exclusion

```c
int count=0, turn=0; /* global vars */
bool interested[2]={false, false}; /* global array */

interleave () {
    pthread_t th0, th1;
    pthread_create(&th0,0,test,0);
    pthread_create(&th1,0,test,1);
    pthread_join(th0,0);
    pthread_join(th1,0);
    printf("%d\n", count);
}

test (int i) {
    for (int j=0; j<MAX; j++) {
        interested[i]=true; turn=i;
        while (interested[1-i]==true && turn==i);
        count= count+1; /* critical section */
        interested[i]=false;
    }
}
```

Hardware Supported Solution

- Challenge so far was designing a solution assuming instruction-set supported only load and store
- If reading and writing can be done in one instruction, designing solutions is much easier
- A popular instruction: test-and-set-lock

```
TSL X, L: register, L: memory loc (bit)
L's content are loaded into X, and L is set to 1
```

Test-and-set seems simple, but can be used to implement complex synchronization schemes

Lock with TSL

```
inter_region:
    TSL REGISTER,LOCK | copy lock to register and set lock to 1
    CMP REGISTER,#0  | was lock zero?
    JNE enter_region  | if it was non-zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0      | store a 0 in lock
    RET | return to caller
```
Hardware Instructions

- MIPS -- Load-Linked/Store Conditional (LL/SC)
- Pentium -- Compare and Exchange, Exchange, Fetch and Add
- SPARC -- Load Store Unsigned Bit (LDSTUB) in v9
- PowerPC -- Load Word and Reserve (lwarx) and Store Word Conditional (stwcx)

Busy Waiting

- Both TSL and Peterson’s require that a process keep looping and checking to see if it can acquire a lock
- Busy waiting wastes CPU cycles
- Better to block and run when ready