Operating Systems

Practical Session 5
Synchronization
Motivation

• Multiprocessing needs some tools for managing shared resources
  – Printers
  – Files
  – Data Bases
Conditions for a good Solution

1. **Mutual Exclusion** – No two processes are in the **critical section** (CS) at the same time

2. **Deadlock Freedom** – If processes are trying to enter the CS, **one** will eventually enter it

3. **Starvation Freedom** – When a process tries to enter its CS, it will eventually succeed
Solution Archetypes

• Busy wait
  – Wastes CPU
  – Priority inversion with busy waiting (*):
    Task \( L \) (low priority) runs and gains exclusive use of resource \( R \).
    \( H \) (high priority) task is introduced and attempts to acquire \( R \)
    and must therefore wait. Note that since \( H \) is busy waiting it
    may still be scheduled (its state remains runnable).
    Result: \( L \) can’t run and release \( R \) because \( H \) is of higher priority
    and is scheduled to run before it. \( H \) on the other hand can’t
    proceed past its busy wait loop.
    Deadlock!

• Sleep & Wake up
  – Also prone to priority inversion but will not deadlock.
    Can you think of a scenario?
Peterson’s Solution

Process = 0

```c
int interested[2];
int barrier;
interested[0]=interested[1]=FALSE

void enter_region(...){
    int other = 1;
    interested[0] = TRUE;
    barrier = 0;
    while (barrier == 0 &&
        interested[1] == TRUE);
}

void leave_region(...){
    interested[0] = FALSE;
}
```

Process = 1

```c
int interested[2];
int barrier;
interested[0]=interested[1]=FALSE

void enter_region(...){
    int other = 0;
    interested[1] = TRUE;
    barrier = 1;
    while (barrier == 1 &&
        interested[0] == TRUE);
}

void leave_region(...){
    interested[1] = FALSE;
}
```
Consider the following variant of Peterson’s solution where the two red lines are swapped. Does this variant solve the mutual exclusion problem?
Peterson’s Solution

Process = 0

```c
int barrier;
int interested[2];
interested[0]=interested[1]=FALSE

void enter_region(...){
    int other = 1;
    barrier = 0;
    interested[0] = TRUE;
    while (barrier == 0 && interested[1] == TRUE);
}

void leave_region(...){
    interested[0] = FALSE;
}
```

Process = 1

```c
int barrier;
int interested[2];
interested[0]=interested[1]=FALSE

void enter_region(...){
    int other = 0;
    barrier = 1;
    interested[1] = TRUE;
    while (barrier == 1 && interested[0] == TRUE);
}

void leave_region(...){
    interested[1] = FALSE;
}
```
Answer

We will get a mutual exclusion violation:

P1 does barrier:=1;
P0 does barrier:=0;
P0 does interested[0]:=true;
P0 does while(barrier == 0 && interested [1]);  // (#t and #f) ➔ #f
P0 enters the CS.
P1 does interested [1]:=true;
P1 does while(barrier == 1 && interested [0]);  // (#f and #t) ➔ #f
P1 enters the CS.
now both processes are in the critical section.
Question 2

Assume the while statement in Peterson's solution is changed to:

```c
while (barrier != process && interested[other] == TRUE)
```
Assume the while statement in Peterson's solution is changed to:

```
while (barrier != process && interested[other] == TRUE)
```

Describe a scheduling scenario in which we will receive a mutual exclusion violation and one in which we will **NOT** receive a Mutual Exclusion violation.
Peterson's Solution

Process = 0

```c
void enter_region(...){
    int other = 1;
    interested[0] = TRUE;
    barrier = 0;
    while (barrier != 0 && interested[1] == TRUE);
}

void leave_region(...){
    interested[0] = FALSE;
}
```

Process = 1

```c
void enter_region(...){
    int other = 0;
    interested[1] = TRUE;
    barrier = 1;
    while (barrier != 1 && interested[0] == TRUE);
}

void leave_region(...){
    interested[1] = FALSE;
}
```
Answer for Q.2

No mutual exclusion violation:

1. One enters and exits and only afterwards the second receives a time quanta.

2. \( P0 \) – interested[0]=true,  
   \( P0 \) – barrier = 0,  
   \( P1 \)- interested[1]=true,  
   \( P1 \) – barrier =1,  
   \( P1 \) passes while loop and enters CS,  
   \( P0 \) – stuck in while loop.

3. Many more....
Answer for Q.2

Mutual exclusion violation:

Process A executes:
`interested [process] = TRUE', `barrier = process'
and falls through the while loop shown above.

While in the critical section, the timer fires and process B executes:
`interested [process] = TRUE', `barrier = process'
and falls through the while loop shown above.

Both processes are in the critical section.
Dekker’s algorithm

Process = 0

1. bool flag[2] = {false, false};
2. int turn = 0;
3. void enter_region(...){
4. flag[0] = true;
5. while (flag[1]) {
6. if (turn==1) {
7. flag[0] = false;
8. while (turn == 1) ;
9. flag[0] = true;}}}
10. void leave_region(...){
11. turn = 1;
12. flag[0] = false; }

Process = 1

1. bool flag[2] = {false, false};
2. int turn = 0;
3. void enter_region(...){
4. flag[1] = true;
5. while (flag[0]) {
6. if (turn==0) {
7. flag[1] = false;
8. while (turn == 0) ;
9. flag[1] = true;}}}
10. void leave_region(...){
11. turn = 0;
12. flag[1] = false; }
Mutual exclusion for n processes: A tournament tree

A tree-node is identified by: [level, node#]
Mutual exclusion for n processes: A tournament tree

Each process locks all the locks between itself and the root lock. When exiting, it unlocks in the reverse order (LIFO).
Mutual exclusion for n processes: A tournament tree

Q: What happens if the unlocking is done in the same order as locking?
Mutual exclusion for n processes: A tournament tree

If 2 starts waiting on root before 0 unlocks, then

```
void leave_region(...){
    interested[0] = FALSE;
}
```

Makes the root node forget that there is an interested process on the left
Lamport’s Bakery Algorithm

```c
int value[N] = {0, 0,...,0}; /* processes get “waiting numbers” */
int busy[N] = {FALSE,..., FALSE}; /* just a flag... */

void enter_region(int i) /* process i entering.. */
{
    busy[i] = TRUE; /* guard the value selection */
    value[i] = max(value[0], value[1], ..., value[N-1]) + 1; /* LAST in line ... */
    busy[i] = FALSE;
    for(k=0; k < N; k ++){
        while (busy[k] == TRUE); /* wait before checking */
        while((value[k] != 0)
            && (value[k] < value[i] || (value[k] == value[i] && k < i ))); /* wait */
    }
}

void leave_region(int i) {
    value[i] = 0;
}
```
Question 4

Assume we have the following *atomic command*:

```c
void swap(bool *a, bool *b) {
    bool temp = *a;
    *a = *b;
    *b = temp;
}
```

Solve the N processes mutual exclusion problem using the swap command.
bool lock = FALSE;

void enter(int i) {
    bool key = TRUE;
    while (key) {
        swap(&key, &lock);
    }
}

void leave(int i) {
    lock = FALSE;
}

void swap(bool *a, bool *b) {
    bool temp = *a;
    *a = *b;
    *b = temp;
}
void swap(bool *a, bool *b) {
    bool temp = *a;
    *a = *b;
    *b = temp;
}

void leave(int i) {
    The idea:
    Seek the next (chronologically ordered) interested process. If such a process exist, grant it the ability to enter the CS. Otherwise reset the lock so that other processes will be able to enter the CS at a later time.
}

The idea:
Use a local variable “key” which will be set to true. Let all processes swap it with current lock and compete over this line. Only the first process to grab it will have a value of false to the key. This process will enter the CS.

void enter(int i) {
    bool lock = FALSE;
    bool waiting[N] = {FALSE, FALSE, ..., FALSE};

    The idea:
    Use a local variable “key” which will be set to true. Let all processes swap it with current lock and compete over this line. Only the first process to grab it will have a value of false to the key. This process will enter the CS.

    waiting[i] = TRUE;
    bool key = TRUE;
    while (TRUE) {
        swap(&key, &lock);
        if (!waiting[i] || !key)
            break;
    }
}

31
Semaphores

• Enable simple synchronization
• An interface of atomic functions supplying mutual exclusion
• Programmers don’t need to bother with synchronization algorithms
Counting semaphore

**Init(i)**

\[
S = i
\]

**down(S)** [the ‘p’ operation]

If \( S \leq 0 \)

the process is blocked.

It will resume execution only after it is woken-up

Else

S--

**up(S)** [the ‘v’ operation]

If (there are blocked processes)

wake-up one of them

Else

S++

- Semaphore’s interface doesn’t enforce the implementation of starvation freedom.
- All operations are ATOMIC! That is, up(s) is more than just \( s := s + 1 \)
- There is no way to access the semaphore’s internal value. Any attempt to access it is a mistake.
- Always remember to initialize the semaphore
XV6 - Spinlock

spinlock.h

// Mutual exclusion lock.
struct spinlock {
    uint locked;     // Is the lock held?
    // For debugging:
    char *name;      // Name of lock.
    struct cpu *cpu; // The cpu holding the lock.
    uint pcs[10];   // The call stack (an array of program counters)
                     // that locked the lock.
};
XV6 - Spinlock

XV6 uses an atomic x86 operation called xchg:

```c
int xchg(int *addr, int value) {
    int temp = *addr;
    *addr = value;
    return temp;
}
```
spinlock.c

// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");

    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
    {
        __sync_synchronize();

        // Tell the C compiler and the processor to not move loads or stores
        // past this point, to ensure that the critical section's memory
        // references happen after the lock is acquired.
        __sync_synchronize();

        // Record info about lock acquisition for debugging.
        lk->cpu = mycpu();
        getcallerpcs(&lk, lk->pcs);
    }
}

release(struct spinlock *lk)
{
    if(!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
    lk->cpu = 0;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that all the stores in the critical
    // section are visible to other cores before the lock is released.
    // Both the C compiler and the hardware may re-order loads
    // and
    // stores; __sync_synchronize() tells them both not to.
    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    asm volatile("movl $0, %0" : "m" (lk->locked) :);

    popcli();
}
XV6 - Scheduler

### proc.c

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns. It loops, doing:
// - choose a process to run
// - swtch to start running that process
// - eventually that process transfers control
//   via swtch back to the scheduler.

void scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;
    for(;;){
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;

            // Switch to chosen process. It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;

            swtch(&(c->scheduler), p->context);
            switchkvm();

            // Process is done running for now.
            // It should have changed its p->state before coming
            // back.
            c->proc = 0;
        }
        release(&ptable.lock);
    }
}
```
Homework
Question 5a – midterm 2010

Let A be an \( n \) processes (finite) mutual exclusion algorithm which is applied with the following guarantee: whenever A is used, any process will attempt to access the critical section only once.

Prove or disprove the following claim:

A is a starvation free algorithm \( \text{if and only if} \) it is a deadlock free algorithm.
The first part of the proof is trivial: starvation freedom also implies deadlock freedom. What about the second one?
Assume that there are $n$ processes and that $A$ is free of deadlocks. Now, let us falsely assume that $A$ is not a starvation free algorithm and contradict this assumption.
If $A$ is not a starvation free algorithm, then there must exist some process $p_i$ which goes through infinitely many steps but still can’t enter the CS.
Answer Q.5a, continued

While $p_i$ goes through its steps other processes manage to enter the CS (we assumed deadlock freedom). As each process $p_j$ may only enter the CS once, at some point all $n-1$ processes will go through (in and out of) the CS and we will be left with $p_i$. At this point, $p_i$ is the only process attempting to enter the CS. Following our initial assumption, there can be no deadlocks and $p_i$ will enter the CS. This contradicts our false assumption that A is not a starvation free algorithm.

QED
Consider the following simple algorithm similar to that shown in class:

1. `int lock` initially 0
2. `boolean interested[n]` initially {false,...,false}

Code for process $i \in \{0,...,n-1\}$

3. `interested[i] := true`
4. `await ((test-and-set(lock)=0) OR (interested[i] = false))`
5. **Critical Section**
6. $j := (i+1) \mod n$
7. `while (j ≠ i) AND (interested[j] = false)`
8. $j := (j+1) \mod n$
9. `if (j=i)`
10. `lock := 0`
11. `else`
12. `interested[j] := false`
13. `interested[i] := false`

**Test-and-set(w)**
do atomically

`prev := w`
`w := 1`
`return prev`

Is this algorithm deadlock free? Prove or disprove by providing an accurate scenario which leads to a deadlock.
The following scenario results in a deadlock:

1. A process $p_0$ is executed and executes all lines up until the beginning of line 13. (Note that ‘lock’ is freed)

2. Another process $p_1$ is executed and allowed to run all the way through the CS. At this point $p_1$ iterates through the other processes and identifies that ‘interested[0]=true’.

3. As a result $p_1$ resets the value ‘interested[0]:=false’ (line 12) and does not free the lock. It then proceeds to line 13 and exits.

4. $p_0$ runs line 13.

5. At this point the system is in a state of deadlock. Any process attempting to enter the CS will be stopped by line 4.