ELEC 377 – Operating Systems

Week 2 – Class 3

Last Class

- Direct vs Indirect Communication
- Synchronization & Buffering
- Threads
- Started Synchronization

Next Week

- Quiz #1 on Tuesday
  - ◊ Covers everything up to the end of Monday’s class
  - ◊ Covers material from first week

Notes

- Labs:

You need a C reference Manual

- you will be programming in C in 4th year, and after you graduate

- good investment

- Douglas Library - QA 76.73 .C

Today

- Synchronization

Process Synchronization

- Most Important Part of the Course and Text

- Concurrent access to shared resources

◊ data inconsistency

◊ need some mechanism to control access to shared resources

Synchronization Example

Account Deposit

```
    …
    account = account + deposit
    …
```

Account Withdrawl

```
    …
    account = account - withdrawl
    …
```

Synchronization Example

Account Deposit	+$ 100
…	
`mov account, reg1`	$ 5,243
`add deposit, reg1`	1 $ 5,343 INT
`move reg1, account`	3 $ 5,343
…	
Account Withdrawl	-$ 100
…	
`mov account, reg1`	$ 5,243
`sub withdrawl, reg1`	2 $ 5,143 INT
`move reg1, account`	4 $ 5,143
…	

Process Synchronization

- Race Condition

◊ Several process handle shared resources

◊ Final value depends on who finishes first

- To prevent race conditions, concurrent processes must be synchronized

◊ train signaling problem

Critical Sections

Account Deposit

```
…
mov account, reg1
add deposit, reg1
move reg1, account
…
```

critical section

Account Withdrawl

```
…
mov account, reg1
sub withdrawl, reg1
move reg1, account
…
```

critical section

Critical Sections

- several processes competing for access to some shared data
- The sections of code where the shared data is accessed and/or modified is called a critical section (each process has its own critical section[s])

- Problem:

◊ Only ***one*** process is allowed in its critical section at a time.

Critical Sections - Requirements

- **Mutual Exclusion** - only one
- **Progress** - if there is no process in a critical section, and more than one process want to enter their critical section, then the selection of a process cannot be postponed indefinitely
- **Bounded Waiting** - once a process is waiting, the other processes can only enter and leave a bounded number of times (no starvation)

Critical Sections - General Model

```
do {
    entry section

                   critical section

    exit section

                   remainder section

} while (1);
```

Two Processes - Algorithm 1

- Shared Variables

  int turn (initially 0)

- Process P_i

```
do {
    while (turn != i);
        critical section
    turn = j;
        remainder section
} while (1);
```

Two Processes - Algorithm 1

- turn = 0, process 0 enters critical section
- process 1 is waiting to enter its critical section
- process 0 leaves critical section, turn = 1
- turn = 1, process 1 enters critical section
- process 1 leaves critical section, turn = 0
- Evaluation:

◊ Mutual Exclusion: only one process in critical section at a time √

◊ bounded waiting - the processes alternate √

Two Processes - Algorithm 1

- turn = 0, process 0 enters critical section
- process 1 is waiting to enter its critical section
- process 0 leaves critical section, turn = 1
- turn = 1, process 1 enters critical section
- process 1 leaves critical section, turn = 0
- Evaluation:

◊ Mutual Exclusion: only one process in critical section at a time √

◊ bounded waiting - the processes alternate √

◊ progress - turn = 0, process 1 is waiting to enter its critical section, process 0 is outside of the critical section but in an infinite loop.

Two Processes - Algorithm 2

- Shared Variables
  boolean flag[2] (both initially false)

- Process P_i

```
do {
  flag[i] = true;
  while (flag[j]);
                        critical section
  flag[i] = false;
    remainder section
} while (1);
```

- flag is used to indicate if the process is waiting for or in the critical section

Two Processes - Algorithm 2

- flag[0] = false, flag[1] = false
- process 0 - flag[0] = true, flag [1] = false , enter critical section
- process 1 - flag [1] = true, flag[0] = true, start looping
- process 0 - leave critical section, flag[0] = false
- process 1, flag[0] = false, stop looping, enter critical section
- process 1 – leave critical section

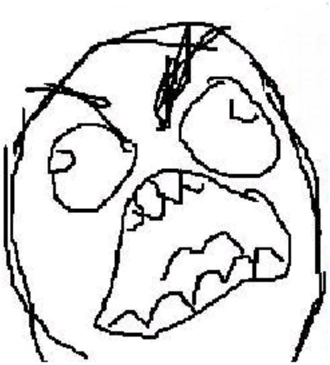
Satisfies mutual exclusion √

Two Processes - Algorithm 2

- flag[0] = false, flag[1] = false
- process 0 - flag[0] = true, flag [1] = false , enter critical section
- process 1 - flag [1] = true, flag[0] = true, start looping
- process 0 - leave critical section, flag[0] = false
- process 1, flag[0] = false, stop looping, enter critical section
- process 1 – leave critical section

Two Processes - Algorithm 2

- flag[0] = false, flag[1] = false
- process 0 - flag[0] = true, **interrupt**
- process 1 - flag [1] = true, flag[0] = true, start looping
- process 0 - flag[1] = true, start looping



- Both processes in infinite loop, neither get to enter critical section, therefore no progress
- Entry to critical section contains a race condition
- Entry to critical section contains a critical section

Two Processes - Algorithm 3

- Combine Alg 1 and Alg 2
- Process P_i

```
do {
  flag[i] = true;
  turn = j
  while (flag[j] and turn = j);
                critical section
  flag[i] = false;
    remainder section
} while (1);
```

- Meets all three requirements - Peterson's Solution

Two Processes - Algorithm 3

- flag[0] = false, flag[1] = false, turn = 0
- process 0 - flag[0] = true, turn = 1, flag[1] = false: enter critical section
- process 1 - flag[1] = true, turn = 0, flag[0] = true: start looping
- process 0 - exit critical section, flag[0] = false
- process 1 - flag[0] = false: enter critical section

Two Processes - Algorithm 3

- flag[0] = false, flag[1] = false, turn = 0
- process 0 - flag[0] = true, turn = 1, interrupt!!
- process 1 - flag[1] = true, turn = 0, flag[0] = true: start looping
- process 0 flag[1] = true, but turn = 0: enter critical section

Two Processes - Algorithm 3

- flag[0] = false, flag[1] = false, turn = 0
- process 0 - flag[0] = true, interrupt!!
- process 1 - flag[1] = true, turn = 0, flag[0] = true: start looping
- process 0 - turn = 1, flag[1] = true:
  start looping
- process 1 - flag[0] = true, but turn = 1: enter critical section
- Mutual Exclusion, Progress, Bounded Waiting are all satisfied!!!
- Works for 2 processes, how about 3 or more?

n Processes - Bakery Algorithm

- Not in Version 7 or 8 of Textbook (But we cover it anyway)
- Based on pick a number in Bakery, Deli's, Government offices.
- Pick the next number (smallest number goes first)

- **Problem**: picking the number

- Numbers are monotonic increasing (1,2,3,3,4,5,5,…)

- numbers not unique

◊ tie goes process with lowest PID.

n Processes - Bakery Algorithm

```
do {
  choosing[i] = true;
  num[i] = max(num[0],..,num[n]) + 1
  choosing[i] = false;
  for (j = 0; j < n; j++){
    while(choosing[j]);
    while(num[j] != 0 &&
          ((num[j],j)<(num[i],i)))    *
  }
    critical section
  num[i] = 0;
    remainder section
} while(1);
```