Concurrency: Mutual Exclusion and Synchronization

Concurrency

Regards: Sharing or competing of resources among multiple processes

Arises because of:

- Multiple applications
- Structured applications programmed as sets of concurrent processes
- Operating system structure—often implemented as the above

Basic requirement: Enforcement of Mutual Exclusion

Concurrency Terms

- Critical Session– code that requires access to shared resource in an exclusive way
- Deadlock(livelock)—more processes do not change state (always change state) because awaiting (of the state of) the others
- Mutual exclusion—when a critical state is reached and resources are accessed, no other critical state depending on those resources is executed
- Race condition—the final state of a shared resource depends on the timing of the changes by a group of processes
- Starvation—a runnable process is always overlooked by the scheduler (does never proceed)

Difficulties of Concurrency

- Sharing of global resources—the order of the access becomes critical
- Operating system managing the allocation of resources optimally—risk of deadlock
- Difficult to locate programming errors—results can be non deterministic and reproducible

Currency

- Communication among processes
- Sharing resources
- Synchronization of multiple processes
- Allocation of processor time

A Simple Example

```
/* reads input from keyboard and outputs it on
  screen */
void echo()
{
  // chin and chout are characters
  1. chin = getchar();
  2. chout = chin;
  3. putchar(chout);
}
```

A Simple Example

```
/* reads input from keyboard and outputs it on
  screen */
void echo()
{
  // chin and chout are characters
  1. chin = getchar();
  2. chout = chin;
  3. putchar(chout);
}
```

Question: what happens if A1, B1—3, A2—3?

Solution: “lock” the whole echo() procedure

Race Condition

- Example with one variable: P1&P2 share the variable a;
  - P1: a = 1;
  - P2: a = 2;
  - P1, P2 → (a, 2) != (a, 1) ← P2, P1
- Example with two variables: P3&P4 sharing variables (b,1) and (c,2)
  - P3: b = b + c
  - P4: c = b + c
  - P3, P4: (b, 3), (c, 5)
  - P4, P3: (c,3), (b, 4)
- Conclusion: the race looser wins!

Operating System Concerns

- Keep track of various processes
- Allocate and deallocate resources
  - Processor time
  - Memory
  - Files
  - I/O devices
- Protect data and resources of each process
- Output of process must be independent of the speed of execution of other concurrent processes

Ways in which processes interact

- Processes unaware of each other
  - Relationship: competition
  - Problems: Mutual Exclusion, DeadLock, Starvation
- Processes indirectly aware of each other (share something)
  - Relationship: cooperation by sharing
  - Problems: ME, DL, Starv, Data coherence
- Process directly aware of each other (have communication primitives)
  - Relationship: Cooperation by communication
  - Problems: DL, Starvation (no ME! Why?)

Competition Among Processes for Resources

- Leave the state of recourses unaffected
- Try to not slow-down processes
- Mutual Exclusion
  - Critical sections
    - Only one program at a time is allowed in its critical section
    - Example: only one process at a time is allowed to send command to the printer
- But we want to avoid:
  - Deadlock (two processes and two resources)
  - Starvation (among three one always looses)

Requirements for Mutual Exclusion

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation

Requirements for Mutual Exclusion

- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Mutual Exclusion: Hardware Support

- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Processor is limited in its ability to interleave programs
  - Multiprocessing
    - disabling interrupts on one processor will not guarantee mutual exclusion

Mutual Exclusion: Hardware Support

- Special Machine Instructions
  - Performed in a single instruction cycle
  - Access to the memory location is blocked for any other instructions

Mutual Exclusion: Hardware Support

```
boolean testset (int *bolt) {
    if (*bolt == 0) {
        *bolt = 1;
        return true;
    }
    else {
    // bolt == 1
    return false;
    }
 }
```

```
const int n = X; // proc. nr
int bolt;
void P(int i){
  …...................
  /* critical section stuff */
  …...........
   /* remainder */

}

void main (){
bolt = ?;
parbegin (P(1),P(2), …, P(n));
}
```

Mutual Exclusion: with Test&Set Hardware Support

```
const int n = X; // number of processes
int bolt;
void P(int i){
  while (true){
      // while bolt == 1 do nothing
      while (!testset(*bolt));
      /* critical section stuff */
      bolt = 0;
      /* remainder */
   }
}

void main (){
   bolt = 0;
   parbegin (P(1), P (2), …, P(n));
}
// wins the first that enters testset with (bolt, 0)
```

Mutual Exclusion: Hardware Support

- Compare&Swap instruction

```
int compare_and_swap
  (int* bolt, int testval, int newval{
     int oldval = *bolt;
     if (oldval == testval)
        *bolt = newval;
     return oldval;
  } // returns the old value of bolt
```

Mutual Exclusion: with Test&Set Hardware Support

```
const int n = X; // number of processes
int bolt;
void P(int i){
  while (true){
      // while bolt == 1 do nothing
      while (compare_and_swap(*bolt, 0, 1) == 1);
      /* critical section stuff */
      bolt = 0;
      /* remainder */
   }
}

void main (){
   bolt = 0;
   parbegin (P(1), P (2), …, P(n));
}
// wins the first that enters c&s with (bolt, 0)
```

Mutual Exclusion: Hardware Support

- Exchange Instruction

```
void exchange(int register, int memory) {
    int temp = memory;
    memory = register;
    register = temp;
}
```

Mutual Exclusion

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi;
   while (true)
   {
      keyi = 1;
      while (keyi != 0)
            exchange (keyi, bolt);
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

Mutual Exclusion Machine Instructions

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections (one bolt variable per session)

Mutual Exclusion Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting. (old elevator effect!)
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs it, the higher priority process will obtain the processor to wait for the critical region

Mutual Exclusion: SW Approach

Assumptions:

- No Hardware support
- Processes share the same memory
- A global variable **turn** is checked and its value dictates who's next
- Processes adopt busy waiting

ME, SW Approach: 1st attempt

- P0:

```
......
   while (turn != 0);
   // do nothing
   /* critical section */
   turn = 1;
```

- P1:

```
......
   while (turn != 1);
   // do nothing
   /* critical section */
   turn = 0;
```

ME, SW Approach: 1st attempt

- P0:

```
......
   while (turn != 0);
   // do nothing
   /* critical section */
   turn = 1;
```

- P1:

```
......
   while (turn != 1);
   // do nothing
   /* critical section */
   turn = 0;
```

Problems:

- turn stores only 1 state!
- Processes **must** alternate
- Speed dictated by the slowest
- If one fails, the other is blocked

ME, SW Approach: 2nd attempt

- Shared variable:
  - boolean flag[2] = {false, false}

- P0:

```
......
1. while (flag[1]);
   // do nothing
2. flag[0] = true;
3. /* critical section */
4. flag[0] = false;
```

- P1:

```
......
1. while (flag[0]);
   // do nothing
2. flag[1] = true;
3. /* critical section */
4. flag[1] = false;
```

ME, SW Approach: 2nd attempt

- P0:

```
......
1. while (flag[1]);
   // do nothing
2. flag[0] = true;
3. /* critical section */
4. flag[0] = false;
```

- P1:

```
......
1. while (flag[0]);
   // do nothing
2. flag[1] = true;
3. /* critical section */
4. flag[1] = false;
```

Problems:

- If a process fails just after setting the flag to true the other is blocked
- Is not independent of the relative process execution speeds => does not guarantee ME

ME, SW Approach: 3rd attempt

- P0:

```
......
1. flag[0] = true;
2. while (flag[1])
   /* do nothing */;
3. /* critical section */
4.flag[0] = false;
```

- P1:

```
......
1. flag[1] = true;
2. while (flag[0]);
   /* do nothing */;
3. /* critical section */
4. flag[1] = false;
```

ME, SW Approach: 3rd attempt

- P0:

```
......
1. flag[0] = true;
2. while (flag[1])
   /* do nothing */;
3. /* critical section */
4. flag[0] = false;
```

- P1:

```
......
1. flag[1] = true;
2. while (flag[0])
   /* do nothing */;
3. /* critical section */
4. flag[1] = false;
```

Properties:

- Again: if a process fails within its critical section, the other is blocked;
- ME is guaranteed
- Processes check their flags independently of what the others do => Risk of deadlock (both processes set the flag to true…)

ME, SW Approach: 4th attempt

- P0:

```
......
1. flag[0] = true;
2. while (flag[1]){
3.   flag [0] = false;
     // delay
4.   flag[0] = true;
   }
5. /* critical section */
6. flag[0] = false;
```

- P1:

```
......
1. flag[1] = true;
2. while (flag[0]){
3.   flag [1] = false;
     // delay
4.   flag[1] = true;
   }
5. /* critical section */
6. flag[1] = false;
```

ME, SW Approach: 4^{th} attempt

- P0:

```
......
1. flag[0] = true;
2. while (flag[1]){
3.   flag [0] = false;
     // delay
4.   flag[0] = true;
   }
5. /* critical section */
6. flag[0] = false;
```

- P1:

```
......
1. flag[1] = true;
2. while (flag[0]){
3.   flag [1] = false;
     // delay
4.   flag[1] = true;
   }
5. /* critical section */
6. flag[1] = false;
```

Properties:

- ME is guaranteed
- But: there is risk for livelock from “mutual courtesy”
  - P0:1, P1:1, P0:2, P1:2, P0:3, P1:3, P0:4, P1:4…..
- Idea: insist on the turn!

ME, SW Approach: Deker's Algorithm

- P0:

```
......
while (true){
  flag[0] = true;
  while (flag[1]){
    if (turn == 1){
      flag [0] = false;
      while (turn == 1)
      /* do nothing */;
      flag[0] = true;
    }
   }
   /* critical section */
   turn = 1;
   flag[0] = false;
}
```

- P1:

```
......
while (true){
  flag[1] = true;
  while (flag[0]){
    if (turn == 0){
      flag [1] = false;
      while (turn == 0)
      /* do nothing */;
      flag[1] = true;
    }
   }
   /* critical section */
   turn = 0;
   flag[1] = false;
}
```

ME, SW Approach: Pearson's Alg.

- P0:

```
......
while (true){
  flag[0] = true;
  turn = 1;
  while (flag[1]&&turn)
      /* do nothing */;
  /* critical section */
  flag[0] = false;
}
```

- P1:

```
......
while (true){
  flag[1] = true;
  turn = 0;
  while (flag[0]&&!turn)
      /* do nothing */;
  /* critical section */
  flag[1] = false;
}
```

- If P0 sets flag to true, P1 cannot enter critical section.
- If P1 is in critical section, flag[1] == true & P0 cannot enter;
- P0 blocked in the while loop (flag[1] is true and turn is 1)
  - P1 is not interested in entering its critical section (impossible; flag[1] == 1)
  - P1 is waiting for its critical section (impossible; turn = 1)
  - P1 is using its critical section repeatedly (impossible! P1 has to set turn to 0)