Concurrent Processes

Operating Systems must…

- The OS must keep track of active processes.
- The OS must allocate and deallocate resources.
  - Processor time
  - Memory
  - Files
  - I/O devices
- The OS must protect the data and physical resources.
- The results of a process must be independent of the speed of execution relative to the speed of other concurrent processes.

An Example - Data Coherence

```
static int a = 1, b = 1;
void P1()                          void P2()
{       a = a + 1;                 {       b = 2 * b;
        b = b + 1;                         a = 2 * a;
}                                  }
```

Process P1	Process P2	a	b
...	...	1	1
a = a + 1	...	2	
...	b = 2 * b		2
b = b + 1	...		3
...	a = 2 * a	4	

Possible Execution Sequences

```
static int a = 1, b = 1;
void P1()                     void P2()
{      s1: a = a + 1;         {      t1: b = 3 * b;
       s2: b = b + 1;                t2: a = 3 * a;
}                             }
```

```
 s1 s2 t1 t2: a = 6 b = 6;       s1 t1 s2 t2: a = 6 b = 4
 s1 t1 t2 s2: a = 6 b = 4;       t1 s1 s2 t2: a = 6 b = 4
 t1 s1 t2 s2: a = 6 b = 4;       t1 t2 s1 s2: a = 4 b = 4
```

With no synchronization, results are typically not deterministic nor reproducible.

Atomic: instruction

Thread T

T: { x++; }

t1: LOD R1, x
t2: ADD R1, 1
t3: STO R1, x

Thread S

S: { x++; }

s1: LOD R1, x
s2: ADD R1, 1
s3: STO R1, x

Possible execution sequences:

(1) t1, t2, t3, s1, s2, s3
(2) t1, t2, s1, t3, s2, s3
(3) t1, t2, s1, s2, t3. s3
(4) t1, t2, s1, s2, s3, t3
(5) t1, s1, t2, t3, s2, s3
(6) t1, s1, t2, s2, t3, s3
(7) t1, s1, t2, s2, s3, t3
(8) t1, s1, s2, t2, t3, s3
(9) t1, s1, s2, t2, t3, s3…

A CR(Critical Region) is an atomic sequence of program segment whose execution must not be interrupted, i.e., must be executed mutual exclusively.

Process/Thread Interaction

Degree of Awareness	Relationship	Process Interaction	Potential Control Problems
They unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock • Starvation
They indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock • Starvation • Data coherence
They directly aware of each other (communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Resource Competition

- Mutual Exclusion
  - Critical resource – a single non-sharable resource.
  - Critical section – portion of the program that accesses a critical resource.
- Deadlock
  - Each process owns a resource that the other is waiting for.
  - Two processes are waiting for communication from the other.
- Starvation
  - A process is denied access to a resource, even though there is no deadlock situation.

Solution to Critical-Section Problem

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and there exist some processes waiting the critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.(**no starvation**)

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i

```
do {
    entry section
        critical section
    exit section
        reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn == i** $\Rightarrow P_i$ can enter its critical section
- Process P_i

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

- Satisfies mutual exclusion, but not progress

First attempt: // must be interleaved

int turn=0;

process 0

```
do{
while(turn!=0);
 critical section
 turn=1;
  remainder section
 }while(1) ;
```

process 1

```
do{
 while(turn!=1);
   critical section
   turn=0;
   remainder section
 }while(1);
```

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] == true** $\Rightarrow$ P_i ready to enter its critical section
- Process P_i

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

- how to set_and_check_flag?

Second attempt: // no protection

int flag[2]={0,0};

process 0

```
do{
        while(flag[1]);
        flag[0] = 1;
        critical section
        flag[0] = 0;
        remainder section
} while(1);
```

process 1

```
do{
        while(flag[0]);
        flag[1] = 1;
        critical section
        flag[1] = 0;
        remainder section
} while(1);
```

Third attempt: // no progress, both are blocked

```
int flag[2]={0,0};
```

process 0

```
do{
 flag[0] = 1;
 while(flag[1]);
   critical section
 flag[0] = 0;
   remainder section
}while(1);
```

process 1

```
do{
 flag[1] = 1;
 while(flag[0]);
   critical section
 flag[1] = 0;
   remainder section
}while(1);
```

Algorithm 3

- Combined shared variables of algorithms 1 and 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);
```

- solves the critical-section problem for two processes.

Peterson algorithm:

```
int flag[2]={0,0}, turn;
```

process 0

```
while(1)
{ flag[0] = 1;
  turn = 1;
  while(flag[1]&& turn==1) ;

  critical section
   flag[0] = 0;
   remainder section
}
```

process 1

```
while(1)
 {  flag[1] = 1;
     turn = 0;
     while(flag[0]&& turn==0) ;

     critical section
      flag[1] = 0;
   remainder section
 }
```

Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,4,5...

Bakery Algorithm

- Notation <≡ lexicographical order (ticket #, process id #)
  - (*a*,*b*) <(*c*,*d*) if $a < c$ or if $a == c$ and $b < d$
  - max (a_0,…, a_{n-1}) is a number, *k*, such that $k \geq a_i$ for $i = 0,\ldots, n - 1$
- Shared data

boolean choosing[n];

int number[n];

Data structures are initialized to **false** and **0** respectively

Bakery Algorithm

```
do { // ith process
    choosing[i] = true;
    number[i] = max(number[0], number[1], …,
                        number[n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++)
      {while (choosing[j]) ;
       while ((number[j] != 0) && (number[j],j) <
                                   (number[i],i)) ;
       }
       critical section
       number[i] = 0;
       remainder section
} while (1);
```

Wait while someone else is choosing

Could have overflow if we don't reset all numbers periodically!

Software Solutions…

- Turns
- Bakery algorithm works
  - Proof by exhaustive cases
  - Bakery algorithm is a pain, complex, difficult to follow
- Other software solutions
  - Dekker's algorithm
  - Peterson's algorithm
- Can hardware help?
- An atomic instruction support for exclusion

Hardware Solutions

- Interrupt disabling
  - not all interrupts need to be disabled
  - might not work in multiprocessor environment
- Special machine instructions
  - test and set instruction (**`TSET`**)
  - exchange instruction (**`XCHG`**)
  - advantages
    - works on any number of processors sharing memory
    - simple and supports multiple critical sections
  - disadvantages
    - busy waiting is employed
    - starvation and deadlock are possible

Test-and-Set

```
bool testset(int &i)
{  if (*i == 0)
   {  *i = 1;
      return false;
   }
   else return true;
}
```

An atomic action
An Instruction

- Shared variable *b* (initialized to 0)
- Only the first P_i who sets *b* enters CS

int b;

testset(&b)

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

```
do {
   while (testset(&lock)) ; // busy waiting
      critical section
   *lock = false;
      remainder section
}
```

Synchronization Hardware

- Atomically swap two variables.

```
void swap(boolean &a, boolean &b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

  boolean lock;

- Process P_i

```
do {
    key = true;
    while (key == true) swap(&lock,&key);
        critical section
    lock = false;
        remainder section
}
```