Operating Systems

Process Synchronization

Cooperating Processes and Shared Data

- A **cooperating process** can affect or be affected by other processes.
- Cooperating processes may directly share address space (e.g. threads share address space).
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires synchronization mechanisms to ensure the orderly execution of cooperating processes.

Recall the Bounded Buffer problem

- Producer/Consumer problem: Producer writes to a buffer and the Consumer reads from the buffer.

  E.g. cat filename | lpr

- Shared-memory solution to bounded-buffer problem (Chapter 4) allows at most *n* – 1 items in buffer at the same time.
- A solution, where all *n locations in the* buffer are used is not simple.
  - Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer Producer

- Producer process

```
item nextProduced;

while (1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Bounded-Buffer Consumer

- Consumer process

```
item nextConsumed;

while (1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

Accessing count concurrently

- What happens when the statements
  counter++;
  counter--;
  are performed concurrently?

- Execution of counter++ in machine code:
  register1 = counter
  register1 = register1 + 1
  counter = register1

- Execution of counter-- in machine code:
  register2 = counter
  registe2 = register2 - 1
  counter = register2

Data inconsistency

Suppose counter initially set to **5**. Execution of **`counter++`** and **`counter--`** consecutively <u>should</u> leave the value at **5**.

Concurrent execution <u>could</u> leave inconsistent data:

- T0: Producer: **`register1 = counter`** *(register1 is 5)*
- T1: Producer: **`register1 = register1+1`** *(register1 is 6)*
- T2: Consumer: **`register2 = counter`** *(register2 is 5)*
- T3: Consumer: **`register2 = register2-1`** *(register2 is 4)*
- T4: Consumer: **`counter = register2`** *(counter is 4)*
- T5: Producer: **`counter = register1`** *(counter is 6)*

Could end up with counter value of 4, 5 or 6!

There is no way to predict the relative speed of process execution, so you cannot guarantee that one will finish before the other.

Atomic Operations

- The statements

  counter++;
  counter--;

  must be performed *atomically*.

- **Atomic operation** means an operation that completes in its entirety without interruption.

Concurrency Problem at Program Execution level

Concurrency problems can arise at the program execution level.
Example: Suppose two processes, P1 and P2 share variables a and b.

```
P1:
        a = a + 1;
        b = b + 1;
P2:
        b = b * 2;
        a = a * 2;
```

Assume a = 7, b = 2 initially.
Assume atomic program instruction execution.

Question: What are the possible ending values for a and b if P1 and P2 execute concurrently?

Concurrency Problem at Program Execution level Continued

Question: *What are the possible ending values for* **`a`** *and* **`b`** *if* **`P1`** *and* **`P2`** *execute concurrently?*

```
a = a + 1;
b = b + 1;
b = b * 2;
a = a * 2;
a == 16
b ==  6
```

```
a = a + 1;
b = b * 2;
b = b + 1;
a = a * 2;
a == 16
b ==  5
```

```
a = a + 1;
b = b * 2;
a = a * 2;
b = b + 1;
a == 16
b ==  5
```

```
b = b * 2;
a = a + 1;
b = b + 1;
a = a * 2;
a == 16
b ==  5
```

```
b = b * 2;
a = a + 1;
a = a * 2;
b = b + 1;
a == 16
b ==  5
```

```
b = b * 2;
a = a * 2;
a = a + 1;
b = b + 1;
a == 15
b ==  5
```

Cannot get

```
a == 15
b ==  6
```

Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

Definitions

- **Concurrent Processes:** Process executions overlap in time.
- **Cooperating processes:** Processes that affect each other during execution (e.g. Parent waits for child; parent communicates with child).
- **Critical Section (CS):** Segment of code that only one process can be in at a time (e.g. segment of code that accesses shared data).
- **Mutual exclusion:** If a process is in its CS, then no other process can be in the same CS. Each CS access must be mutually exclusive (mutex).
- **Atomic execution:** Execution of code that is not interrupted.
- **Busy waiting:** Repeated execution of a code loop while waiting for an event.
- **Deadlock:** when two or more processes are permanently blocked.
- **Starvation:** When a process is indefinitely delayed; other processes are given the resource this process needs.

The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

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 that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting**. (no starvation) A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the *n* processes.

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

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

Satisfies *mutual exclusion*, but not *progress*. *It requires strict alternation of processes in CS.*

If turn ==0, P_1 cannot enter critical section even though P_0 might be in remainder section. (If P_0 does not need a turn again, P_1 is forced to wait.)

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

- Satisfies mutual exclusion, but not progress requirement. . (It even seems worse, critically dependent on timing.) For example:

$\mathbf{P_0}$ *sets* **flag[0]** *to* **true** *and* $\mathbf{P_1}$ *sets* **flag[1]** *to* **true**

Now both processes loop forever, have DEADLOCK!

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);
```

- Meets all three requirements; solves the critical-section problem for two processes.

Does Algorithm 3 Meet Requirements

Mutual Exclusion? YES

P_i can only enter critical section

***if* `flag[j]==false` *or when* `turn==i`**

Progress and bounded waiting? YES

P_i can only be prevented from entering critical section

if stuck in the while loop with

`flag[j]==true` *and when* `turn==j`

if P_j is not ready to enter critical section, then

`flag[j]==false`

if P_j is ready to enter critical section, then

`flag[j]==true` *and then either*

`turn==i` *OR* `turn==j` *SO*

either P_i or P_j will enter critical section.

***if P_j enters critical section, it will reset* `flag[j]` *to* `false`**

at end and P_i will be able to enter critical section.

Bakery Algorithm

Critical section for n processes

- The Bakery algorithm works like a bakery (or other shop) where customers take a number as they enter the shop and are served in order of the numbers they hold.
- 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., 1,2,3,3,3,3,4,5...

Bakery Algorithm

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

  boolean choosing[n]; //true while process choosing a number

  int number[n]; //number assigned to each process

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

Bakery Algorithm

```
do {
    choosing[i] = true;        //Process i is choosing a number
                               //Process i gets next number
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;       //Process i done choosing
    for (j = 0; j < n; j++) {  //Wait for processes with lower numbers
            while (choosing[j]) ;
            while ((number[j] != 0) && (number[j],j) < (number[i],i)) ;
    }
        critical section
    number[i] = 0;  //When done, set number back to zero
        remainder section
} while (1);
```

//This algorithm satisfies mutual exclusion, progress and bounded waiting.

Synchronization Hardware

- Hardware features can make synchronization easier.
- If the system has a single processor, then one option is to disable interrupts during the critical section. This will not work for multiple processors and it is inefficient.
- Many systems provide special instructions that are executed atomically.
  - *Either* test memory word and set value
  - *Or* swap contents of two memory words
- We can use these special instructions to solve the critical section problem.

Synchronization Hardware

- TestAndSet allows the program to test and modify the content of a word atomically

.

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;

    return rv;
// return value as tested, but set it to true; because, if
// it was false before, you snag it, so others can't have it

}
```

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

```
do {
    while (TestAndSet(lock)) ;  //acquire
lock
        critical section
    lock = false;                //release
lock
        remainder section
} while(1);
```

Notes on TestAndSet

- The previous algorithm satisfies mutual exclusion.
- It does not satisfy bounded waiting. Starvation is possible.
- The algorithm works for multiple processes. (But there is no ordering of processes waiting to enter the CS).

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
} while (1);
```

//Satisfies mutual exclusion, but not bounded wait.

- Solution for no starvation: Shared data (initialized to **false**):
  boolean lock;
  boolean waiting[n]; //list of processes waiting for CS
- Process P_i

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = TestAndSet(lock);
    waiting[i] = false;
        critical section
    j = (i + 1)%n;
    while((j != i) && !waiting[j])
        j = (j + 1)%n;
    if (j == i )
        lock = false;
    else
        waiting[j] = false;
    remainder section
} while (1);
```

Satisfying Requirements

- The previous algorithm satisfies the three requirements
- Mutual Exclusion: The first process to execute TestAndSet(lock) when lock is false, will set lock to true so no other process can enter the CS.
- Progress: When a process exits the CS, it either sets lock to false, or waiting[j] to false, allowing the next process to proceed.
- Bounded Waiting: When a process exits the CS, it examines all the other processes in the waiting array in a circular order. Any process waiting for CS will have to wait at most n-1 turns.