Process Coordination

Why is it needed?

- Processes may need to share data
  - More than one process reading/writing the same data (a shared file, a database record,…)
  - Output of one process being used by another
  - Needs mechanisms to pass data between processes
- Ordering executions of multiple processes may be needed to ensure correctness
  - Process X should not do something before process Y does something etc.
  - Need mechanisms to pass control signals between processes

Interprocess Communication (IPC)

- Mechanism for processes P and Q to communicate and to synchronize their actions
  - Establish a communication link
- Fundamental types of communication links
  - Shared memory
    - P writes into a shared location, Q reads from it and vice-versa
  - Message passing
    - P and Q exchange messages
- We will focus on shared memory, will discuss issues with message passing later

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Producer Consumer Problem

- Paradigm for cooperating processes
- *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.
- Basic synchronization requirement
  - Producer should not write into a full buffer
  - Consumer should not read from an empty buffer
  - All data written by the producer must be read exactly once by the consumer

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

- We will see how to create such shared memory between processes in the lab

Bounded-Buffer: Producer Process

```
item nextProduced;

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

Bounded-Buffer: Consumer Process

```
item nextConsumed;

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

- The solution allows at most *n* – 1 items in buffer (of size *n*) at the same time. A solution, where all *n* buffers are used is not simple
- Suppose 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

Shared data

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

Will this work?

Producer process

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

Consumer process

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

The Problem with this solution

- The statement "*counter++*" may be implemented in machine language as:

  register1 = counter

  register1 = register1 + 1

  counter = register1

- The statement "*counter--*" may be implemented as:

  register2 = counter

  register2 = register2 – 1

  counter = register2

- If both the producer and consumer attempt to update *counter* concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.

An Illustration

- Assume *counter* is initially 5. One interleaving of statements is:

  producer: *register1 = counter (register1 = 5)*
  producer: *register1 = register1 + 1 (register1 = 6)*
  consumer: *register2 = counter (register2 = 5)*
  consumer: *register2 = register2 – 1 (register2 = 4)*
  producer: *counter = register1 (counter = 6)*
  consumer: *counter = register2 (counter = 4)*

- The value of *counter* may be either 4 or 6, where the correct result should be 5.

Race Condition

- A scenario in which the final output is dependent on the relative speed of the processes
  - Example: The final value of the shared data *counter* depends upon which process finishes last
- Race conditions must be prevented
  - Concurrent processes must be synchronized
  - Final output should be what is specified by the program, and should not change due to relative speeds of the processes

Atomic Operation

- An operation that is either executed fully without interruption, or not executed at all
  - The "operation" can be a group of instructions
    - Ex. the instructions for *counter++* and *counter--*
- Note that the producer-consumer problem's solution works if *counter++* and *counter--* are made atomic
- In practice, the process may be interrupted in the middle of an atomic operation, but the atomicity should ensure that no process uses the effect of the partially executed operation until it is completed

The Critical Section Problem

- *n* processes all competing to use some shared data (in general, use some shared resource)
- Each process has a section of code, called critical section, in which a 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, irrespective of the relative speeds of the processes
- Also known as the Mutual Exclusion Problem as it requires that access to the critical section is mutually exclusive

Requirements for Solution to the Critical-Section Problem

Mutual Exclusion: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

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.

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.

Entry and Exit Sections

- Entry section: a piece of code executed by a process just before entering a critical section
- Exit section: a piece of code executed by a process just after leaving a critical section
- General structure of a process P_i

.

.

entry section

critical section

exit section

remainder section /*remaining code */

- Solutions vary depending on how these sections are written

Petersen's Solution

- Only 2 processes, P_0 and P_1
- Processes share some common variables to synchronize their actions
  - *int turn = 0*
    - *turn = i* $\Rightarrow$ P_i 's turn to enter its critical section
  - *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;
    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
  - Can be extended to *n* processes by pairwise mutual exclusion – too costly

Solution for *n* Processes: Bakery Algorithm

- 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 <≡ 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 $k \geq a_{\text{i}}$ for i - 0,

    …, $n - 1$
- Shared data

    boolean choosing[n];

    int number[n];

 Data structures are initialized to *false* and 0 respectively

Bakery Algorithm

```
do {
    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);
```

Hardware Instruction Based Solutions

- Some architectures provide special instructions that can be used for synchronization

- TestAndSet: Test and modify the content of a word atomically

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

- *Swap*: *Atomically* swap two variables.

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

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

```
do {
    while (TestAndSet(lock)) ;
        critical section
    lock = false;
        remainder section
}
```

Mutual Exclusion with Swap

- Shared data (initialized to *false*):

```
boolean lock;
boolean waiting[n];
```

- Process P_i

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

Semaphore

- Widely used synchronization tool
- Does not require busy-waiting
  - CPU is not held unnecessarily while the process is waiting
- A Semaphore *S* is
  - A data structure with an integer variable *S.value* and a queue *S.q* of processes
  - The data structure can only be accessed by two atomic operations, *wait(S)* and *signal(S)* (also called *P(S)* and *V(S)*)
- Value of the semaphore S = value of the integer *S.value*

wait and *signal* Operations

wait (S): *if (S.value > 0) S.value --;*
else {
add the process to S.q;
block the process;
}

signal (S): *if (S.q is not empty)*
choose a process from S.q and unblock it
else S.value ++;

- Note: which process is picked for unblocking may depend on policy. Also, implementations can make *S.value < 0* also (change *wait* and *signal* code appropriately)

Solution of *n*-Process Critical Section using Semaphores

- Shared data:

  semaphore mutex; / initially mutex = 1 */*

- Process P_i:

```
do {
  wait(mutex);
       critical section
  signal(mutex);
         remainder section
  } while (1);
```

Ordering Execution of Processes using Semaphores

- Execute statement *B* in P_j only after statement *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

- Multiple such points of synchronization can be enforced using one or more semaphores

Pitfalls

- Use carefully to avoid
  - Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
  - Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

Example of Deadlock

- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
⋮	⋮
signal(S);	signal(Q);
signal(Q)	signal(S);

Two Types of Semaphores

- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement.
- Counting semaphore – value can be any positive integer
  - Useful in cases where there are multiple copies of resources
  - *l-exclusion* problem: at most *l* processes can be in their critical section at the same time
- Can implement a counting semaphore using a binary semaphore easily (do it yourself)

Internal Implementations of Semaphores

- How do we make *wait* and *signal* atomic?
  - Should we use another semaphore? Then who makes that atomic? ☺
- Different solutions possible
  - Interrupts:
    - Disable interrupts just before a *wait* or a *signa*l call, enable it just after that
    - Works fine for uniprocessors, but not for multiprocessors
  - Use s/w-based or h/w-instruction-based solutions to put entry and exit sections around *wait/signal* code
    - Since *wait/signal* code is of small size, won't busy wait for too long

Classical Problems of Synchronization

- Bounded-Buffer Producer-Consumer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1

Bounded-Buffer Problem: Producer Process

```
do {
        …
    produce an item in nextp
        …
    wait(empty);
    wait(mutex);
        …
    add nextp to buffer
        …
    signal(mutex);
    signal(full);
} while (1);
```

Bounded-Buffer Problem: Consumer Process

```
do {
    wait(full)
    wait(mutex);
        …
    remove an item from buffer to nextc
        …
    signal(mutex);
    signal(empty);
        …
    consume the item in nextc
        …
} while (1);
```

Readers-Writers Problem

- A common shared data
- Reader process – only reads data
- Writer process – only writes data
- Synchronization requirements
  - Writers should have exclusive access to the data
    - No other reader or writer can access the data at that time
  - Multiple readers should be allowed to access the data if there is no writer accessing the data

Solution using Semaphores

<u>Shared data</u>

semaphore mutex, wrt;

Initially

mutex = 1, wrt = 1, readcount = 0

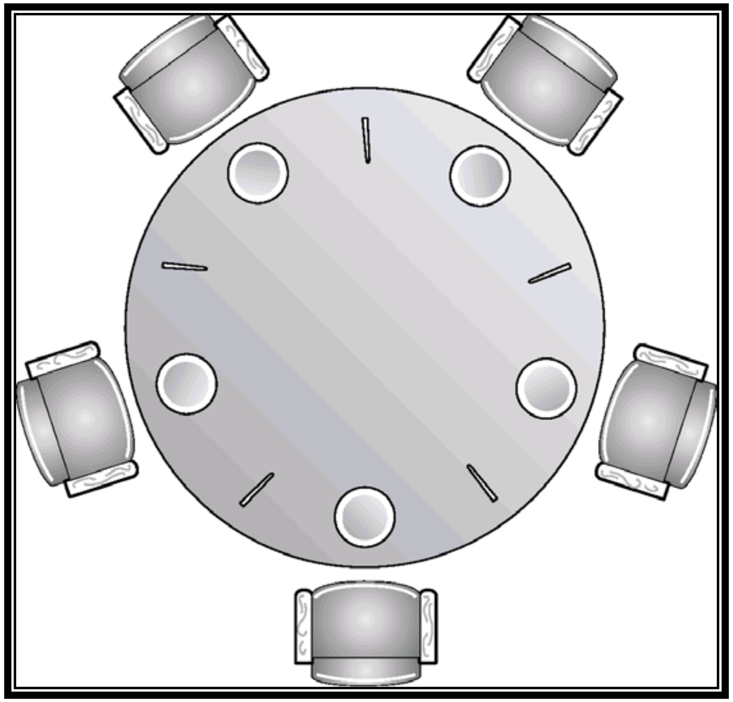
<u>Writer</u>

```
wait(wrt);
        …
        perform write
         …
signal(wrt);
```

<u>Reader</u>

```
wait(mutex);
   readcount++;
   if (readcount == 1)
       wait(wrt);
signal(mutex);
 …
perform read
…
wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
signal(mutex):
```

Dining-Philosophers Problem



- Shared data

semaphore chopstick[5];

Initially all values are 1

Dining-Philosophers Problem

- Philosopher *i*:

```
do {
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
      …
      eat
      …
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
      …
      think
      …
} while (1);
```

Other Synchronization Constructs

- Programming constructs
  - Specify critical sections or shared data to be protected by mutual exclusion in program using special keywords
  - Compiler can then insert appropriate code to enforce the conditions (for ex., put wait/signal calls in appropriate places in code)
- Examples
  - Critical regions, Monitors, Barriers,…