Process Interaction Through Cooperation

- We shall study the following related topics:
  - memory sharing, also referred as critical section problem, situations when two or more processes access shared data
  - synchronization, situations in which progress of one process depends upon the progress of another process,
  - inter-process communication, where processes communicate through message exchange.

- Later, we shall also discuss multithreaded processes, as extension of single-threaded processes, a type of processes we have been studying.

Principles of Process Concurrency

- In a single-processor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution. Even though actual parallelism is not achieved and there is overhead in switching between processes, interleaved executions provides major benefits in processing efficiency.

- In a multi-processor systems, it is possible not only to interleave the execution of processes but also to overlap them.

- Although it might seem that interleaving and overlapping present different problems, both techniques can be viewed as examples of concurrent processing, and both present the same problems.

A Simple Example

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Shared Variable</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.1) (y = x)</td>
<td>(x = 5)</td>
<td>(2.1) (z = x)</td>
</tr>
<tr>
<td>(1.2) (y = y + 1)</td>
<td>(z = z + 2)</td>
<td></td>
</tr>
<tr>
<td>(1.3) (x = y)</td>
<td>(2.3) (x = z)</td>
<td></td>
</tr>
</tbody>
</table>

- Scenario 1: \((1.1), (1.2), (1.3), (2.1), (2.2), (2.3)\) \(\Rightarrow x = 8\)
- Scenario 2: \((1.1), (1.2), (2.1), (2.2), (2.3), (1.3)\) \(\Rightarrow x = 6\)

Scenario 2 leaves shared variable \(x\) in inconsistent state.

The problem occurs when one process reads the shared variable, then a second process reads it before the first one writes it out. Thus, each process will increment the same value and one increment will be lost.
In the case of single-processor system, the reason we have a problem is that an interrupt can stop instruction execution anywhere in a process.

In the case of multi-processor system, we have the same condition and, in addition, a problem can be caused because two processes may be executing simultaneously by two processors and both trying to access the same shared variable.

But solutions to both types of problem are identical:
- A mechanism to ensure the orderly access to shared data by cooperating processes to maintain data consistency.

In general, the solution to this problem is not to allow any other process to access the shared data (variable) while one process is using it.

The section of the process's program that uses the shared data will be called the critical section of the process. Each process sharing some variables has a corresponding critical section where its code uses the shared memory.

What we require to solve, called the critical section problem (C.S.P), is to ensure that only one process at a time will be executing in its critical section.

Requirements for C.S. Problem Solution
1. Mutual Exclusion: If process \( P \) 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 sections, then the selection of the process that will enter the critical section next cannot be postponed indefinitely; and, only processes wishing to enter their critical sections (and process exiting its critical section, if any) can participate in the selection.
3. Bounded Waiting: 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.
- Assumed that each process executes at a nonzero speed and no assumption concerning relative speed of the processes.
- Requirement No. 1. is necessary, while 2. and 3. are desirable.

Comment on Process Structure
- Here is a general structure of typical process \( P_i \), as given in the textbook, when critical section problem discussed:

```c
    do {
        entry section
        critical section
        exit section
        reminder section
    } while (1);
```
- The entry and exit sections are enclosed in boxes to highlight these important segments of code.
- Note that infinite looping is not necessary, since a process may go through its critical section limited number of times.
There are (at least) four general ways how critical section problem can be solved:

- **algorithmic approaches:** leave complete responsibility to processes; these approaches are likely to have high processing overhead and the because of their complexity the risk of logical error is significant. E.g. Dekker’s algorithm.
- **use of special-purpose machine instructions.** E.g. disable/enable interrupt instructions or Intel bts instruction
- **use of some level of support from operating system.** E.g. semaphores and operation on semaphores.
- **use of some level of support from programming languages.** E.g. monitors.

**Solutions for Critical Section Problem**

- **Problems with this solution:**
  - disable/enable interrupt instructions are privileged instructions, thus user processes are not able to execute them,
  - it can’t be used to support multiple critical sections,
  - it does not work with multiprocessor systems.

**Semaphore can only be accessed via two atomic, i.e. indivisible, operations wait and signal:**
- **wait** (semaphore *S); defined as:
  ```c
  S->value = S->value - 1;
  if (S->value < 0)
  { add this process to S->list and block (); // i.e. put issuing process in blocked state
  }
  ```
- **signal** (semaphore *S); defined as:
  ```c
  S->value = S->value + 1;
  if (S->value <= 0)
  { remove process P from S->list and wakeup (P);
    //i.e. put P in ready state
  }
  ```

**Semaphore is defined as:**
- `typedef struct {
  int value;
  struct process *list;
} semaphore;`
- `semaphore *S;`
  - **S->value must be initialized,** to some non-negative integer.
  - **List S->list is initially empty.**

**Then, a solution for the critical section problem is to:**
- define a semaphore, call it `mutex`, with its integer initialized to 1,
- add the appropriate operations on the semaphore in the boxes.
Two Process One Way Synchronization

- Semaphores are also very efficient synchronization mechanism.
- Problem: Execute code B in process P2 only after code A executed in process P1
  
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>wait(flag)</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
</tbody>
</table>

- Define semaphore flag with its integer initialized to 0.
- This problem is referred as "Two Process One Way Synchronization", and it can be stated as follows:
  - each of two processes (P1 and P2) has a special (synchronization) point in its code,
  - process P2 may cross its synchronization point only when P1 crosses or reaches its synchronization point,
  - process P1 may cross its synchronization point unconditionally.

Two Process Two Way Synchronization

- Processes P1 and P2 interact the following way. Each process has a special (synchronization) point in its code, and:
  - P1 may cross its synchronization point only when P2 crosses or reaches its synchronization point,
  - P2 may cross its synchronization point only when P1 crosses or reaches its synchronization point.

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>signal(flag)</td>
<td>wait(flag)</td>
</tr>
</tbody>
</table>

- Define semaphores x and y both initialized to 0.

3 Process 3 Way Synchronization

- Processes P0, P1 and P2 interact such that each process has a special (synchronization) point in its code, and any of processes has to wait for other two to cross or reach their synchronization points, and only then that process may cross its synchronization point.
- Define semaphores S0, S1 and S2 all initialized to 0.

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>signal(S0);</td>
<td>signal(S1);</td>
<td>wait(S1);</td>
</tr>
<tr>
<td>signal(S0);</td>
<td>signal(S1);</td>
<td>wait(S1);</td>
</tr>
<tr>
<td>wait(S2);</td>
<td>wait(S2);</td>
<td>signal(S2);</td>
</tr>
<tr>
<td>wait(S0);</td>
<td>wait(S0);</td>
<td>signal(S2);</td>
</tr>
<tr>
<td>signal(S1);</td>
<td>signal(S1);</td>
<td>wait(S1);</td>
</tr>
<tr>
<td>wait(S0);</td>
<td>wait(S0);</td>
<td>signal(S1);</td>
</tr>
<tr>
<td>wait(S2);</td>
<td>wait(S2);</td>
<td>wait(S1);</td>
</tr>
<tr>
<td>wait(S0);</td>
<td>wait(S0);</td>
<td>wait(S1);</td>
</tr>
</tbody>
</table>

3 Process 3 Way Synchronization (cont.)

- Here is another solution for the same problem:
- Define semaphores S0, S1 and S2 all initialized to 0.

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>signal(S1);</td>
<td>signal(S0);</td>
<td>wait(S0);</td>
</tr>
<tr>
<td>signal(S2);</td>
<td>signal(S0);</td>
<td>wait(S0);</td>
</tr>
<tr>
<td>wait(S0);</td>
<td>signal(S2);</td>
<td>wait(S0);</td>
</tr>
<tr>
<td>wait(S0);</td>
<td>signal(S0);</td>
<td>wait(S0);</td>
</tr>
<tr>
<td></td>
<td>signal(S1);</td>
<td>signal(S0);</td>
</tr>
<tr>
<td></td>
<td>wait(S1);</td>
<td>signal(S0);</td>
</tr>
<tr>
<td></td>
<td>wait(S2);</td>
<td>wait(S2);</td>
</tr>
<tr>
<td></td>
<td>wait(S2);</td>
<td>wait(S2);</td>
</tr>
</tbody>
</table>

- Note: One of two solutions may have a problem. Which one? Why?
There are two processes: Producer and Consumer, and one N slot buffer.

In each iteration, Producer first produces an item and then attempts to put the item into a slot of the buffer. If all slots of the buffer are full, Producer has to wait for an empty slot. If and when there is an empty slot, the item is stored and Producer goes into the next iteration.

In each iteration, Consumer first attempts to take an item from a slot in the buffer. If there is no an item in the buffer, Consumer has to wait for an item to arrive. If and when an item is available, Consumer take an item and consumes it. When done, Consumer starts the next iteration. Items should be taken in the order they are put into the buffer.

This problem is also referred as “The Bounded-Buffer Consumer-Producer Problem”

There are three improvements in this solution:
- in & out variables could be local
- the critical section problem with the variable counter in statements “counter++” and “counter--”; but we know how to solve it.
- the solution has busy waiting; this is a serious problem.

Why is this solution better than one on the previous slide?
The last solution seems to violate mutual exclusion rules since both processes can be reading and writing the buffer array at the same time. It turns out however that this will not cause a problem. The structure of solution ensures that they will never be accessing the same element of the buffer array at the same time. But this logic is tricky and applies only to arrays. In general, it is very easy to make mistakes in formulating synchronization solutions. Additionally, it is often very difficult to do testing just running processes a number of times. The safest course is to be very careful and conservative. But do not over-synchronize, i.e. do not unnecessary block a process, to loose advantages of concurrent executions of processes.

Comments on Semaphore Usage

- Semaphores used for mutual exclusion (in critical section problem) are often called mutex semaphores and they are always used by a pair of calls, first wait (mutex) and then signal (mutex) and both are made by the same process.
- On the other hand, for synchronization semaphores, wait (synch) and signal (synch) are always called by different processes.
- When you see semaphore solutions to process communication problems it is very useful to distinguish between those two very different uses of semaphores. This helps to understand the solution. Many inter-process communication problems requires both styles of semaphore use.
- In all solutions we assume that no process fails, since if a process fails while holding a semaphore then system will deadlock for sure.

Be very careful when using any synchronization mechanism.

Let S and Q be two semaphores initialized to 1

1. $P_1$ wait(S); 2. $P_2$ wait(Q);
3. $P_1$ signal(Q); 4. $P_2$ signal(S);

Deadlock will happen in this special scenario (and several others):


Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

Note if the order of two waits in the code of only one of processes is changed, deadlock could not happen.

Deadlock Involving Semaphores

typedef struct {
  . . .
} xx;

xx buffer[N];
int in=0, out=0;

Code of Producers:

(xx Prod; while (true)

Code of Consumers:

{ xx Cons;

2 Producers, 2 Consumers & N-slot Buffer

Semaphores: full=0, empty=N,
mutex1=1, and mutex2=1

Semaphores used for mutual exclusion (in critical section problem) are often called mutex semaphores and they are always used by a pair of calls, first wait (mutex) and then signal (mutex) and both are made by the same process.

On the other hand, for synchronization semaphores, wait (synch) and signal (synch) are always called by different processes.

When you see semaphore solutions to process communication problems it is very useful to distinguish between those two very different uses of semaphores. This helps to understand the solution. Many inter-process communication problems requires both styles of semaphore use.

In all solutions we assume that no process fails, since if a process fails while holding a semaphore then system will deadlock for sure.
A race condition occurs when there is some sort of sharing data in common memory between the two (or more) processes in such way that final result depends on the order of execution of instructions in the multiple processes.

Example:
- Processes P1 and P2 share global variables b and c, with initial values b=1 and c=2.
- At some point in its execution, P1 executes statement b=b+c
- At some point in its execution, P2 executes statement c=b+c
- If P1 executes its statement first, the final values: b=3 & c=5
- If P2 executes its statement first, the final values: b=4 & c=3
- And either set of the final values is valid.

Distinguish between critical section problem (which ends with values in inconsistent state) and race condition.

Race Condition

Readers - Writers Problem

The solution gives priority to readers, i.e. no reader is kept waiting unless a writer has already obtained permission to use shared object.

Shared data: int readcount = 0;
Semaphores: mutex = 1, wrt = 1;

Writer:
wait(wrt);
wait(mutex);
/*writing is performed*/
... signal(mutex);
wait(mutex);
wait(mutex);
/*reading is performed*/
...
wait(mutex);
readcount--;
if (readcount == 0)
signal(wrt);
signal(mutex);

— Starvation of writers possible.

Dining-Philosophers Problem

Semaphores chopstick [i], i=0, 1, 2, 3, 4, initialized to 1
The structure of Philosopher i (i = 0,1,2,3,4):

While (true) {
  wait (chopstick[i]);
  wait (chopstick[ (i + 1) % 5]);
  // eat
  signal (chopstick[i]);
  signal (chopstick[ (i + 1) % 5]);
  // think
}

But this solution has deadlock!

Dining-Philosophers Problem: Solution 1

Take left chopstick (ch i)
wait (chopstick[i]);
wait (chopstick[ (i + 1) % 5]);
// eat
signal (chopstick[i]);
signal (chopstick[ (i + 1) % 5]);
// think

Take right chopstick (ch i+1)
Release left chopstick
Release right chopstick
Semaphore Implementation

\[\text{wait}(S)\]
\begin{verbatim}
{
    disable interrupts
    S.value = S.value - 1;
    if (S.value < 0)
    {
        add this process to S.L;
        block issuing process;
    }
    enable interrupts;
}
\end{verbatim}

\[\text{signal}(S)\]
\begin{verbatim}
{
    disable interrupts;
    S.value = S.value + 1;
    if (S.value <= 0)
    {
        remove process P from S.L;
        unblock process P;
    }
    enable interrupts;
}
\end{verbatim}

- This is, basically, a solution for this critical section problem appropriate only for implementations in single processor systems.

- In multi-processor systems, type of Intel bts instruction is used for implementation of semaphore operations, i.e. to achieve a solution to this critical section problem for \texttt{wait} and \texttt{signal} semaphore operations.

Properties of Special Instruction Solution

- Advantages:
  - supports single and multi-processor systems
  - simple and easy to verify
  - it can be used to support multiple critical sections; each critical section can be defined by its own variable (bit)

- Disadvantages (in general):
  - busy waiting is employed
  - starvation is possible
  - deadlock is possible: Scenario: Lower priority process P1, already in its critical section, is interrupted by higher priority process P2, and P2 attempts to enter its critical section. P2 will forever be in a busy waiting loop, but at the same time will prevent P1 to get any CPU time.

  But this is appropriate for implementing semaphore

IPC Through Message Exchange

- IPC = Inter-Process Communication
- Message passing system – processes communicate with each other without shared variables but by sending and receiving messages.
- IPC facility provides two operations:
  - \texttt{send(message)}, i.e. write \texttt{receive(message)}, i.e. read
- If \(P\) and \(Q\) wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via \texttt{send/receive}
Implementation Issues

- 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? Usually variable.
- Is a link unidirectional or bi-directional?
- Two main approaches in message passing:
  - Direct Communication
  - Indirect Communication

Unix Pipes

- Pipes are one of most used Unix process communication mechanisms, and can be classified as indirect communication.
- Pipes are half duplex, i.e. data flows only in one direction.
- A pipe can be used only between processes that have a common ancestor that created the pipe.
- A pipe is created by calling the pipe function:
  ```
  int pipe(int fd[2]);
  Returns: 0 if OK, -1 on error.
  ```
- Two open file identifiers are returned by the pipe system call through the fd argument. fd[0] is open for reading, while fd[1] is open for writing and the output of fd[1] is the input for fd[0].
- Message boundaries are not preserved in a pipe.
- But, a pipe in a single process is useless.

Unix Pipes (cont.)

- The process that calls pipe then creates child process. For a pipe in direction from the parent to the child, the parent closes the read end of the pipe (with close(fd[0])), while the child closes the write end of the pipe (with close(fd[1])).
- The parent then can use the standard write system call with fd[1] as fid, while the child can use the standard read system call with fd[0] as fid.
- After reading all data from the pipe whose write end has been closed, the next read returns 0.
- If there is no data in a pipe whose write end is not closed, the process that issues a pipe read will be blocked until data is written in the pipe.
- If a pipe is full, the process that issues write will be blocked until there is enough room in the pipe for write data to be stored.
- A pipe write into the pipe whose all reads have been closed would cause a fatal (segmentation) program error.