Process Cooperation and Synchronization

- Process synchronization
  
  ... mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

- Why do processes cooperate?
  - modularity: breaking up a system into several sub-systems
  - e.g.: an interrupt handler and device driver that need to communicate
  - convenience: users might want to have several processes share data
  - speedup: a single program is run as several sub-programs

- How do processes cooperate?
  - communication abstraction: producers and consumers
    - producers produce a piece of information
    - consumers use this information
  - abstraction helps deal with general “phenomena” and simplifies correctness arguments

Example of Process Cooperation: Bounded Buffers

Processes communicate through a buffer of finite capacity

N: integer -- buffer size
nextin = nextout = 1 initially; -- start of buffer
buffer: array of size N

Producer:
Repeat
  -- produce an item in tempin
  while (nextin+1) mod n = nextout do waitabit;
  buffer[nextin] := tempin;
  nextin := (nextin+1) mod n;

Consumer:
Repeat
  -- consume the item in tempout
  while nextin = nextout do waitabit;
  tempout := buffer[nextout];
  nextout := (nextout+1) mod n;

- How to store N items?
Bounded Buffers Using Counters

\[\begin{align*}
N: \text{integer} & \quad \text{-- buffer size} \\
counter: \text{integer} & \\
nexin = nexout = 1 \text{ initially} & \quad \text{-- start of buffer} \\
\text{buffer: array of size } N & \\
\end{align*}\]

**Producer:**

Repeat

\[\begin{align*}
\quad & \text{\textit{while} counter = } N \text{ do waitabit;} \\
\quad & \text{buffer}[\text{nexin}] := \text{tempin}; \\
\quad & \text{nexin} := (\text{nexin}+1) \text{ mod } N; \\
\quad & \text{counter} := \text{counter}+1; \\
\end{align*}\]

**Consumer:**

Repeat

\[\begin{align*}
\quad & \text{\textit{while} counter = } 0 \text{ do waitabit;} \\
\quad & \text{tempout} := \text{buffer}[\text{nexout}]; \\
\quad & \text{nexout} := (\text{nexout}+1) \text{ mod } N; \\
\quad & \text{counter} := \text{counter}-1; \\
\quad & \text{-- consume the item in tempout} \\
\end{align*}\]

Interleaving of Increment/Decrement

• Each of increment and decrement are actually implemented as a series of machine instructions on the underlying processor

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>register1 := counter</td>
<td>register2 := counter</td>
</tr>
<tr>
<td>register1 := register1 + 1</td>
<td>register2 := register2 + 1</td>
</tr>
<tr>
<td>counter := register1</td>
<td>counter := register2</td>
</tr>
</tbody>
</table>

• An interleaving

– counter = 5; a producer followed by a consumer

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>register1 := counter</td>
<td>[register1 = 5]</td>
</tr>
<tr>
<td>register1 := register1 + 1</td>
<td>[register1 = 6]</td>
</tr>
<tr>
<td>counter := register1</td>
<td>[counter = 5]</td>
</tr>
</tbody>
</table>

The Problem

• Increment and decrement are not \textit{atomic} or \textit{uninterruptable}
  – two or more operations are executed \textit{atomically} if the result of their execution is equivalent to that of some serial order of execution
  – operations which are always executed atomically are called \textit{atomic}
    • byte read; byte write;
    • word read; word write

• The code containing these operations creates a \textit{race condition}
  – produces inconsistencies in shared data

The Solution

• The producer and consumer processes need to synchronize
  – so that they do \textit{not} access shared variables at the same time
    -- this is called \textit{mutual exclusion}
      • the \textit{shared} and \textit{critical} variables can be accessed one process at a time
      • access must be \textit{serialized} even if the processes attempt \textit{concurrent} access
      • as in the previous example

• General framework for achieving this: \textit{Critical Sections}
  – work independent of the particular context or need for synchronization
Critical Sections

- Critical sections are written as
  
  \[
  \begin{align*}
  &\text{entry-section} \\
  &\text{critical-section-code} \\
  &\text{exit-section}
  \end{align*}
  \]

  - the *entry section* controls access to make sure no more than one process \( P_i \) gets to access the critical section at any given time
    - acts as a *guard*
  - the *exit section* does bookkeeping to make sure that other processes that are waiting know that \( P_i \) has exited

- We will see two examples of mutual exclusion on a *pair* of processes
  - turn counters and array of flags

Two-Process Solutions: Turn Counters

- Shared integer variable: \( \text{turn} \) (initialized to 0)
  
  \[
  \begin{align*}
  &\text{for processes } P_i \text{ and } P_j, P_i \text{ executes:} \\
  &\quad \text{while } (\text{turn} \neq i) \text{ waitabit; } \\
  &\quad \text{CRITICAL SECTION; } \\
  &\quad \text{turn} := j; \\
  \end{align*}
  \]

  - the while loop is the *entry section*
    - process \( P_i \) waits till its turn occurs
  - the single instruction \( \text{turn} := j \) constitutes the *exit section*
    - informs the other process of its turn

- Mutual exclusion?
  - assume atomic loads and stores

- Drawbacks?
  - if \( P_j \) never wants to execute the critical section, \( P_i \) cannot reenter;
    - access must alternate

Two-Process Solutions: Array of Flags

- Boolean array: \( \text{flag} \) (initialized to false)
  
  \[
  \begin{align*}
  &\text{for process } P_i \text{ executes:} \\
  &\quad 1: \text{flag}[i] := \text{true;} \\
  &\quad 2: \text{while flag}[j] \text{ waitabit;} \\
  &\quad \text{CRITICAL SECTION} \\
  &\quad 3: \text{flag}[i] := \text{false;}
  \end{align*}
  \]

- Mutual exclusion?
  - suppose *not*; then \( \exists \) a sequence s.t. both \( P_i \) and \( P_j \) are in the critical section
    - when \( P_i \) entered the critical section,
      - it found \( \text{flag}[j] \) to be false at time \( t_1 \); instruction 2
      - it had set \( \text{flag}[i] \) to true at time \( t_0 (0 < t_1) \)
    - if \( P_j \) was already in the critical section
      - it must have found \( \text{flag}[i] \) to be false at time \( t_3 (3 < t_0) \); instruction 2
      - it must have set \( \text{flag}[j] \) to true at time \( t_2 (2 < t_3) \)
      - so \( P_i \) cannot find \( \text{flag}[j] \) false at time \( t_1 > t_0 > t_2 \)

Is This Good Enough?

- *No*: \( P_i \) and \( P_j \) can be looping on instruction 2 forever
  - both \( P_i \) and \( P_j \) set their flags to true and wait for the other flag to be false
  - leads to deadlock, which will be discussed in detail later

- So, what criteria should a general critical-section solution satisfy?
Criteria for Correctness

Three conditions

• **Mutual exclusion**

• **Progress**
  – at least one process requesting entry to a critical section will be able to enter if there is no other process in it

• **Bounded waiting**
  – no process waits indefinitely to enter the critical section once it has requested entry

Two-Process Solutions: Petersen’s Algorithm

• Combines the previous two ideas
  – process \( P_i \) executes

    1: \( \text{flag}[i] := \text{true} \)
    2: \( \text{turn} := j \)
    3: while (\( \text{flag}[j] \) and (\( \text{turn} == j \))) \( \text{waitabit} \)
    4: \( \text{flag}[i] := \text{false} \)

• Does the algorithm satisfy the three criteria?

Petersen’s Algorithm: Mutual Exclusion

1: \( \text{flag}[i] := \text{true} \)
2: \( \text{turn} := j \)
3: while (\( \text{flag}[j] \) and (\( \text{turn} == j \))) \( \text{waitabit} \)
   CRITICAL SECTION
4: \( \text{flag}[i] := \text{false} \)

• Suppose: \( P_i \) is in its critical section, and \( P_j \) is wanting to enter
• This can happen only if either
  – \( P_i \) found \( \text{flag}[j] \) false, or
  – \( P_i \) found \( \text{turn} == i \)
  – in the first case: \( P_j \) will set \( \text{turn} \) after \( P_i \) did, and find \( \text{turn} == i \)
  – in the second case: \( P_j \) has already set \( \text{turn} == i \)
  – in both cases: \( P_j \) will wait till \( \text{flag}[i] == \text{false} \)

Petersen’s Algorithm: Progress and Bounded Waiting

1: \( \text{flag}[i] := \text{true} \)
2: \( \text{turn} := j \)
3: while (\( \text{flag}[j] \) and (\( \text{turn} == j \))) \( \text{waitabit} \)
   CRITICAL SECTION
4: \( \text{flag}[i] := \text{false} \)

• To prove progress:
  – if \( P_j \) is not ready to enter the critical section
    • \( \text{flag}[j] \) will be false • \( P_i \) can enter

• To prove bounded waiting:
  – let \( P_i \) be in the critical section and \( P_j \) be waiting on instruction 3 above
  – if \( P_i \) exits and goes elsewhere,
    • either \( P_j \) will find \( \text{flag}[i] \) to be false
    • if not, \( P_j \) will attempt to reenter the critical section, setting \( \text{turn} := j \)
    • in either case, \( P_j \) will find the condition for waiting in (3) to be false and will enter the critical section
N-Process Synchronization

- Intuition:
  - Processes ask for a ticket from an agent and get an integer valued ticket
  - Processes are not guaranteed to receive unique tickets
  - A process waits until all processes with smaller ticket values have finished going through the critical region
    - In case of a tie, let the process with the smaller PID go first
    - Leads to a FCFS prioritizing strategy
  - The algorithm is akin to taking a ticket and waiting for a turn in a bakery and is called the bakery algorithm

The Bakery Algorithm

1: choosing[i] := true
2: number[i] := max(number[0], number[1], ..., number[n-1]) + 1
3: choosing[i] := false
4: for j := 0 to n-1
   5:     do begin
   6:        while choosing[j] do waitabit
   7:        while number[j] != 0 and (number[j], j) < (number[i], i) do waitabit
   8:     end
9: CRITICAL SECTION
10: number[i] := 0;

Bakery Algorithm: Mutual Exclusion

- Assumption: No processes fail
- Ticketed processes will exit in FCFS order
- The set of processes is finite
  - Implies that every process will get its turn in finite time

Bakery Algorithm: Progress and Bounded Waiting

• Consider P_i in its critical section, and P_k trying to enter its own
  - 3 cases when P_i executes step 2:
    • Case 1: P_i found choosing[k] true
    • Case 2: P_i found choosing[k] false because P_k had executed step 3.
    • Case 3: P_i found choosing[k] false because P_k had not executed step 1.
  - In cases 1 and 2, P_k would have found ticket[j] non-zero
    • So, it must have found (number[i], i) < (number[k], k)
  - In case 3: P_k would compute ticket[k] > ticket[i]
    • So, P_k will remain stuck on step 7 while P_i is in the critical section
Hardware Support

- Primitives
  - hardware instructions which enable simpler/efficient solutions to common synchronization problems

- Criterion for choosing primitives:
  - universality
    - being able to build arbitrary functionality from simpler units
  - minimising scope
    - don’t want to stop interrupts for whole critical sections
    - on a uniprocessor, disabling interrupts is sufficient to prevent interleaving

- Two common primitives
  - test-and-set
  - swap

Test-and-Set

- Semantics:
  - given boolean variables X, Y, atomically set X := Y; Y := true
    ```plaintext
    function test-and-set(var target:boolean) boolean;
    begin
      test-and-set := target;
      target := true;
    end;
    ```

- Mutual exclusion using test-and-set
  ```plaintext
  lock : boolean := false
  while test-and-set (lock) waitabit
  CRITICAL SECTION
  lock := false
  ```

Test-and-Set: Correctness

- Mutual exclusion
  - first process P_i entering critical section sets lock := true
  - further processes that execute test-and-set don’t enter since test-and-set evaluates to true after (from atomicity) this
  - when P_i exits, lock is set to false, so the next process P_j to execute the instruction will find test-and-set = false and will enter the critical section

- Progress
  - trivially true

- Unbounded waiting
  - possible since depending on the timing of evaluating the test-and-set primitive, other processes can enter the critical section first
  - does not guarantee fairness!

The Swap Primitive

- Semantics: Atomically exchange the values of given variables X and Y
  ```plaintext
  temp = X; X = Y; Y = temp;
  ```

- Swap and mutual exclusion:
  ```plaintext
  function test-and-set(var v: boolean): boolean
  var t := true;
  swap (v, t);
  return t;
  ```
Can We Get Bounded Waiting?

- Introduce a boolean array called \textit{waiting} of size $n$ and a single local boolean variable \textit{key}.

- Entry (we are looking at process $P_i$)
  
  \begin{verbatim}
  waiting[i] := true;
  key := true;
  while ( waiting[i] and key ) do
    key := test-and-set (lock);
    waiting[i] := false
  execute CRITICAL SECTION
  \end{verbatim}

- Exit
  
  - find the next process $j$ that has \texttt{waiting}[$j$] = 1 by stepping through \textit{waiting}
    - set \texttt{waiting}[$j$] := false;
    - process $P_j$ will immediately enter the critical section
    - if no such process exists then, \texttt{lock} := false;
      - this means the next process to come and wait will be the first process and can enter
        the critical section freely

Correctness: Mutual Exclusion

- Every (interested) $P_i$ executes that test&set at least once
- $P_i$ enters the critical region provided:
  - key is false in which case there is no process in the critical region
  - or
    - if it was waiting, because waiting[$i$] was reset to false by the unique
      process that was blocking it in the critical region
- Either of the above events can occur exactly once and hence mutual exclusion

Correctness: Bounded Waiting

- Let us examine: Find the next process $j$ that has \texttt{waiting}[$j$] = 1 by stepping through \textit{waiting}
  - \texttt{waiting}[$j$] := false;
  - Any concern about bounded waiting will mean that there must be a process $P_j$ waiting
  - This means that using the above approach for finding the ``next'' process $P_j$'s turn will arrive in no more than ($n-1$) steps implying that there is a finite bound on the waiting

Lecture Summary

- Process Synchronization
  - background
  - critical sections
  - mutual exclusion algorithms
    - 2-process: Turn, Flag, Petersen
    - N-process: Bakery
  - hardware primitives

- Limitations of these algorithms
  - do not solve more general synchronization problems
    - only mutual exclusion
    - involve "busy-waiting"

- Are there more general solutions?
Next Lecture

- Process Synchronization (contd.)
  - higher-level primitives: semaphores
    - semantics
    - implementation
  - classical problems of synchronization
    - bounded-buffer
    - readers-writers
    - dining-philosophers

Readings
- Silberschatz/Galvin: Sections 6.4 - 6.5