Part II
Process Management
Chapter 6: Process Synchronization
Process Synchronization

- Why is synchronization needed?
- Race Conditions
- Critical Sections
- Pure Software Solutions
- Hardware Support
- Semaphores
- Monitors
- Message Passing
Why is Synchronization Needed? 1/4

int Count = 10;

Process 1       Process 2

Count++; Count--;

Count = ? 9, 10 or 11?
Why is Synchronization Needed? 2/4

```c
int Count = 10;
```

**Process 1**
- LOAD Reg, Count
- ADD #1
- STORE Reg, Count

**Process 2**
- LOAD Reg, Count
- SUB #1
- STORE Reg, Count

The problem is that the execution flow may be switched in the middle!
### Why is Synchronization Needed? 3/4

<table>
<thead>
<tr>
<th>Inst</th>
<th>Reg</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>SUM</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>STORE</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst</th>
<th>Reg</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>SUB</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>STORE</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

*erases the previous value 11*
Why is Synchronization Needed? 4/4

<table>
<thead>
<tr>
<th></th>
<th>Process 1</th>
<th></th>
<th>Process 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst</td>
<td>Reg</td>
<td>Memory</td>
<td>Inst</td>
<td>Reg</td>
</tr>
<tr>
<td>LOAD</td>
<td>10</td>
<td>10</td>
<td>LOAD</td>
<td>10</td>
</tr>
<tr>
<td>ADD</td>
<td>11</td>
<td>10</td>
<td>SUB</td>
<td>9</td>
</tr>
<tr>
<td>STORE</td>
<td>11</td>
<td>11</td>
<td>STORE</td>
<td>9</td>
</tr>
</tbody>
</table>

erases the previous value 9
Race Conditions

- A *Race Condition* occurs, if
  - two or more processes/threads access and manipulate the *same* data *concurrently*, and
  - the outcome of the execution *depends on the particular order* in which the access takes place.

- *Synchronization* is needed to prevent race conditions from happening.

- *Synchronization* is a difficult topic. Don’t miss a class; otherwise, you will miss a lot of things.
Critical Section and Mutual Exclusion

- A critical section is a section of code in which a process accesses shared resources.
- Thus, the execution of critical sections must be mutually exclusive (e.g., at most one process can be in its critical section at any time).
- The critical-section problem is to design a protocol that processes can use to cooperate.

```c
int count; // shared

count++;        count--;       cout << count;
```
A critical section protocol consists of two parts: an entry section and an exit section.

Between them is the critical section that must run in a mutually exclusive way.

```c
while (1);
```
Solutions to the Critical Section Problem

- Any solution to the critical section problem must satisfy the following three conditions:
  - Mutual Exclusion
  - Progress
  - Bounded Waiting
- Moreover, the solution cannot depend on CPU’s relative speed and scheduling policy.
Mutual Exclusion

- If a process $P$ is executing in its critical section, then *no* other processes can be executing in their critical sections.
- The *entry protocol* should be capable of blocking processes that wish to enter but cannot.
- Moreover, when the process that is executing in its critical section exits, the *entry protocol* must be able to know this fact and allows a waiting process to enter.
Progress

- If no process is executing in its critical section and some processes wish to enter their critical sections, then
  - Only those processes that are waiting to enter can participate in the competition (to enter their critical sections).
  - No other process can influence this decision.
  - This decision cannot be postponed indefinitely.
Bounded Waiting

- After a process made a request to enter its critical section and before it is granted the permission to enter, there exists a *bound* on the number of times that other processes are allowed to enter.

- Hence, even though a process may be blocked by other waiting processes, it will not be waiting forever.
Software Solutions for Two Processes

- Suppose we have two processes, $P_0$ and $P_1$.
- Let one process be $P_i$ and the other be $P_j$, where $j = 1 - i$. Thus, if $i = 0$ (resp., $i = 1$), then $j = 1$ (resp., $j = 0$).
- We want to design the enter-exit protocol for a critical section so that mutual exclusion is guaranteed.
Algorithm I: 1/2

Global variable `turn` controls who can enter the critical section.

Since `turn` is either 0 or 1, only one can enter.

However, processes are forced to run in an alternating way.

process $P_i$

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

if it is not my turn, I wait

enter

exit

I am done, it is your turn now
Algorithm I: 2/2

- This solution does not fulfill the progress condition.
- If $P_j$ exits by setting turn to $i$ and terminates, $P_i$ can enter but cannot enter again.
- Thus, an irrelevant process can block other processes from entering a critical section.

process $P_i$

```c
do {
    while (turn != i);
    turn = j;
} while (1);
```

if it is not my turn, I wait

enter
critical section

exit

I am done, it is your turn now

If it is not my turn, I wait

enter
critical section

exit

I am done, it is your turn now

This solution does not fulfill the progress condition.

Thus, an irrelevant process can block other processes from entering a critical section.
Algorithm II: 1/2

```c
bool flag[2];
do {
    flag[i] = TRUE;
    while (flag[j]);
    flag[i] = FALSE;
} while (flag[i]);
```

- Variable `flag[i]` is the “state” of process $P_i$: interested or not-interested.
- $P_i$ expresses its intention to enter, waits for $P_j$ to exit, enters its section, and finally changes to “I am out” upon exit.
The correctness of this algorithm is timing dependent!

If both processes set flag[i] and flag[j] to TRUE at the same time, then both will be looping at the while forever and no one can enter.

Bounded waiting does not hold.
Algorithm III: a Combination 1/4

```c
bool flag[2];  // process P_i
int turn;

do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    flag[i] = FALSE;
} while (/* conditions */);
```

**Critical section**
- **Enter**: I am interested
- **While**: yield to you first
- **Exit**: I am done
- **Wait while you are interested and it is your turn.**

// process Pi
Algorithm III: Mutual Exclusion

If both processes are in their critical sections, then

- \(\text{flag}[j] \land \text{turn} == j\) (\(P_j\)) and \(\text{flag}[i] \land \text{turn} == i\) (\(P_i\)) are both FALSE.
- \(\text{flag}[i]\) and \(\text{flag}[j]\) are both TRUE
- Thus, \(\text{turn} == i\) and \(\text{turn} == j\) are FALSE.
- Since \(\text{turn}\) can hold one value, only one of \(\text{turn} == i\) or \(\text{turn} == j\) is FALSE, but not both.
- We have a contradiction and \(P_i\) and \(P_j\) cannot be in their critical sections at the same time.
Algorithm III: Progress 3/4

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

- If $P_i$ is waiting to enter, it must be executing its `while` loop.
- Suppose $P_j$ is not in its critical section:
  - If $P_j$ is not interested in entering, `flag[j]` was set to `FALSE` when $P_j$ exits. Thus, $P_i$ may enter.
  - If $P_j$ wishes to enter and sets `flag[j]` to `TRUE`, it will set `turn` to $i$ and $P_i$ may enter.
- In both cases, processes that are not waiting do not block the waiting processes from entering.
Algorithm III: Bounded Waiting 4/4

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

- When $P_i$ wishes to enter:
  - If $P_j$ is outside of its critical section, then $\text{flag}[j]$ is FALSE and $P_i$ may enter.
  - If $P_j$ is in its critical section, eventually it will set $\text{flag}[j]$ to FALSE and $P_i$ may enter.
  - If $P_j$ is in the entry section, $P_i$ may enter if it reaches while first. Otherwise, $P_j$ enters and $P_i$ may enter after $P_j$ sets $\text{flag}[j]$ to FALSE and exits.

- Thus, $P_i$ waits at most one round!

process $P_i$
Hardware Support

- There are two types of hardware synchronization supports:
  - Disabling/Enabling interrupts: This is slow and difficult to implement on multiprocessor systems.
  - Special privileged machine instructions:
    - Test and set (TS)
    - Swap
    - Compare and Swap (CS)
Interrupt Disabling

- Because interrupts are disabled, no context switch will occur in a critical section.
- Infeasible in a multiprocessor system because all CPUs must be informed.
- Some features that depend on interrupts (e.g., clock) may not work properly.

```c
do {
    disable interrupts
    critical section
    enable interrupts
} while (1);
```
Special Machine Instructions

- **Atomic**: These instructions execute as one uninterrupted unit. More precisely, when such an instruction runs, all other instructions being executed in various stages by the CPUs will be stopped (and perhaps re-issued later) until this instruction finishes.

- Thus, if two such instructions are issued at the same time, even though on different CPUs, they will be executed sequentially.

- **Privileged**: These instructions are, in general, privileged, meaning they can only execute in supervisor or kernel mode.
The Test-and-Set Instruction

Mutual exclusion is obvious as only one TS instruction can run at a time.

However, progress and bounded waiting may not be satisfied.

```c
bool TS(bool *key)
{
    bool save = *key;
    *key = TRUE;
    return save;
}
```

bool lock = FALSE;

do {
    while (TS(&lock));
    lock = FALSE;
} while (1);

```c
bool TS(bool *key)
{
    bool save = *key;
    *key = TRUE;
    return save;
}
```

bool lock = FALSE;

do {
    lock = FALSE;
} while (1);
Problems with Software and Hardware Solutions

- All of these solutions use *busy waiting*.

- *Busy waiting* means a process waits by executing a tight loop to check the status/value of a variable.

- Busy waiting may be needed on a multiprocessor system; however, it wastes CPU cycles that some other processes may use productively.

- Even though some personal/lower-end systems may allow users to use some atomic instructions, unless the system is lightly loaded, CPU and system performance can be low, although a programmer may “think” his/her program looks more efficient.

- So, we need a better solution.