Silberschatz and Galvin
Chapter 6

Process Synchronization

Topics discussed

- Process synchronization
- Mutual exclusion--hardware
- Higher-level abstractions
  - Semaphores
  - Monitors
- Classical example problems
Process Synchronization

- Process coordination—Multi processor considerations caused by interaction of processes on multiple CPUs operating simultaneously
- Shared state (e.g., shared memory or shared variables)
- When concurrent processes interact through shared variables, the integrity of the variables’ data may be violated if the access is not coordinated

**What is the problem?**

**How is coordination achieved?**

---

Process Synchronization

**Problem Statement**

- Result of parallel computation on shared memory can be nondeterministic
- Example
  
  \[
  A = 1; \parallel A = 2;
  \]
- What is result in A? 1, 2, 3, ...?
- **Race condition**: (race to completion)
  - cannot predict what will happen since the result depends on which one goes faster
  - what happens if both go at exactly the same speed?
Process Synchronization
Example

Assume that X is a bank account balance

- **Process A: payroll**
  - load X, R
  - add R, 1000
  - store R, X

- **Proc B: ATM Withdraw**
  - load X, R
  - add R, -100
  - store R, X

If two processes are executed sequentially, e.g.,

- load X, R
- add R, 1000
- store R, X

...... O.S. context switch

- load X, R
- add R, -100

No problem!

If two processes are interleaved, e.g.,

- load X, R
- add R, -100

...... O.S. context switch

- load X, R
- add R, 1000
- store R, X

...... O.S. context switch

- store R, X

Problem occurs!
Basic Assumptions for system building

- The order of some operations are irrelevant (some operations are independent)
  \[
  A = 1; \quad || \quad B = 2;
  \]
- Can identify certain segments where interaction is critical
- **Atomic operation(s)** must exist in hardware
  - Atomic operation: either happens in its entirety without interruption or not at all
  - Cannot solve critical section problem without atomic operations

Atomic Operations

- Example, consider the possible outcomes of an atomic and a non-atomic `printf`
  ```
  printf("ABC"); \quad || \quad printf("CBA");
  ```
  but `printf` is too big to be atomic (hundreds or thousands of instructions executed, I/O waits, etc.)
- Commonly-found atomic operations
  - memory references
  - assignments on simple scalars (e.g., single bytes or words)
  - operations with interrupts disabled on uniprocessor
- Cannot make atomic operations if you do not have them (but can use externally supplied operations like disk accesses if they are atomic to build more generally-useful atomic operations)
- More on implementation of atomic operations later
Process Coordination

- Lower-level atomic operations are used to build higher-level ones (more later)
  - e.g., semaphores, monitors, etc.
- **Note**: in analysis, no assumption on the relative speed of two processes can be made. Process coordination requires explicit control of concurrency.

Process Coordination Problems

**Producer/Consumer applications**

- **Producer**--creates information
- **Consumer**--uses information
- Example--piped applications in Unix
  ```
  cat file.t | eqn | tbl | troff | lpr
  ```
- **Bounded buffer** between producer and consumer is filled by producer and emptied by consumer
Bounded Buffer

initialize counter, in, out to 0

**Producer**

while (true) {
    produce an item in nextp;
    while counter == n do noop;
    buffer[in] = nextp;
    in = in + 1 mod n;
    counter = counter + 1;
}

**Consumer**

while (true) {
    while counter == 0 do noop;
    nextc := buffer[out];
    out := out + 1 mod n;
    counter := counter - 1;
    consume item in nextc;
}

Bounded Buffer

- Concurrent execution of producer and consumer can cause unexpected results, even if we assume that assignment and memory references are atomic
- For example, interleavings can result in counter value of n, n+1, or n-1 when there are really n values in the buffer

**Producer**

load counter

add 1
store counter

results in a value of n+1

**Consumer**

load counter

subtract 1
store counter
Controlling interaction

- Similar problems even when we are running the *same* code in the two processes
  - Example: shopping expedition
- Need to manage interaction in areas in which interaction is critical
- **Critical section**: section of code or collection of operations in which only one process may be executing at a given time
  - examples: counter, shopping

---

**Critical Section**

```plaintext
Producer
while (true) {
    produce an item in nextp;
    while counter == n do noop;
    buffer[in] = nextp;
    in = in + 1 mod n;
    counter = counter + 1;
}

Consumer
while (true) {
    while counter == 0 do noop;
    nextc := buffer[out];
    out := out + 1 mod n;
    counter := counter - 1;
    consume item in nextc;
}
```

*Critical Sections*
Critical Section

- Critical section operations
  - entry: request permission to enter critical section
  - exit: marks the end of the critical section
- Mutual exclusion: make sure that only one process is in the critical section at any one time
- Locking: prevent others from entering the critical section
  - entry: then is acquiring the lock
  - exit: is releasing the lock

Critical Section

- Solution must provide
  - Mutual exclusion
  - Progress: if multiple processes are waiting to enter the critical section and there is no process in the critical section, eventually one of the processes will gain entry
  - Bounded waiting: no indefinite postponement
  - Deadlock avoidance
    - deadlock example
      - P1 gains resource A; P2 gains resource B;
      - P1 waits for resource B; P2 waits for resource A;
- Next, we will consider a number of potential solutions
Critical Section Solution?
Using a counter to show turn

\[
\text{turn} := 1;
\]

while(true) {
    \textit{non critical stuff}
    while (\text{turn} == 2); /* wait */
    \textit{critical section}
    \text{turn} = 2;
    \textit{non critical stuff}
}

while(true) {
    \textit{non critical stuff}
    while (\text{turn} == 1); /* wait */
    \textit{critical section}
    \text{turn} = 1;
    \textit{non critical stuff}
}

Critical Section Solution?
Check if other process busy

\[
\text{p1busy} := \text{false}; \text{p2busy} := \text{false};
\]

while(true) {
    \textit{non critical stuff}
    while (\text{p2busy}); /* wait */
    \text{p1busy} = \text{true};
    \textit{critical section}
    \text{p1busy} = \text{false};
    \textit{non critical stuff}
}

while(true) {
    \textit{non critical stuff}
    while (\text{p1busy}); /* wait */
    \text{p2busy} = \text{true};
    \textit{critical section}
    \text{p2busy} = \text{false};
    \textit{non critical stuff}
}
Critical Section Solution?

Set flag **before** check

\[
p1\text{busy} := \text{false};\ p2\text{busy} := \text{false};
\]

\[
\begin{aligned}
&\text{while(true) \{} \\
&\text{non critical stuff} \\
&p1\text{busy} = \text{true}; \\
&\text{while (p2\text{busy}); /* wait */} \\
&\text{critical section} \\
&p1\text{busy} = \text{false}; \\
&\text{non critical stuff} \\
&\} \\
\end{aligned}
\]

\[
\begin{aligned}
&\text{while(true) \{} \\
&\text{non critical stuff} \\
&p2\text{busy} = \text{true}; \\
&\text{while (p1\text{busy}); /* wait */} \\
&\text{critical section} \\
&p2\text{busy} = \text{false}; \\
&\text{non critical stuff} \\
&\} \\
\end{aligned}
\]

\[
p1\text{busy} := \text{false};\ p2\text{busy} := \text{false};
\]

Critical Section Solution?

More complicated wait

\[
p1\text{busy} := \text{false};\ p2\text{busy} := \text{false};
\]

\[
\begin{aligned}
&\text{while(true) \{} \\
&\text{non critical stuff} \\
&p1\text{busy} = \text{true}; \\
&\text{while(p2\text{busy}) \{} \\
&\text{p1\text{busy} = false;} \\
&s\text{leep;} \\
&p1\text{busy} = \text{true;} \\
&\} \\
&\text{critical section} \\
&p1\text{busy} = \text{false}; \\
&\text{non critical stuff} \\
&\} \\
\end{aligned}
\]

\[
\begin{aligned}
&\text{while(true) \{} \\
&\text{non critical stuff} \\
&p2\text{busy} = \text{true}; \\
&\text{while(p1\text{busy}) \{} \\
&\text{p2\text{busy} = false;} \\
&s\text{leep;} \\
&p2\text{busy} = \text{true;} \\
&\} \\
&\text{critical section} \\
&p2\text{busy} = \text{false}; \\
&\text{non critical stuff} \\
&\} \\
\end{aligned}
\]
Mutual exclusion solution requirements

- Mutual exclusion is preserved
- The progress requirement is satisfied
- The bounded-waiting requirement is met

Critical Section Solution?
Peterson’s Algorithm (Alg. 3)

```plaintext
while(true) {
    non-critical stuff
    p1busy = true;
    turn = 2;
    while (p2busy and turn == 2) ; /* wait */
    critical section
    p1busy = false;
    non critical stuff
}
```

```plaintext
while(true) {
    non-critical stuff
    p2busy = true;
    turn = 1;
    while (p1busy and turn == 1) ; /* wait */
    critical section
    p2busy = false;
    non critical stuff
}
```
Mutual Exclusion Hardware Implementation

- Implementation requires atomic hardware operation
- For example, test-and-set
  
  ```
  function test-and-set(var target:boolean): boolean;
  begin
    test-and-set = target;
    target = true;
  end;
  ```
- Sample use
  ```
  lock = false;
  miscellaneous processing...
  while(true) {
    while(test-and-set(lock)) do ;
    critical section
    lock = false;
  }
  ```

Mutual Exclusion in Hardware

- Can also use other atomic operations (swap, etc.) to implement if test-and-set is not available
- What are the problems?
  - Hardware dependent. Different hardware requires different implementation
  - Hard to generalize to more complex problems
  - Inefficient because of busy wait
- In general, would prefer to use an abstraction, which could be implemented once for each hardware architecture.
- **Semaphores**: one such abstraction
Semaphores

- Defined by Dijkstra (1965)
- Two operations, P(s) and V(s). s is the **semaphore**, a non-negative integer
- **P**: from the Dutch *probern* (to test) represents a *wait*
  - P(s)--decrement s by 1 if possible (i.e., without going negative). If s==0 then wait until it is possible to decrement s without going negative.
- **V**: from the Dutch *verhogen* (to increment) represents a *signal*
  - V(s)--increment s by 1 in a single atomic action

Mutual Exclusion using Semaphores

```c
mutex = 1;  /* mutex is the semaphore */
miscellaneous processing...
while (true) {
    P(mutex);
    critical section
    V(mutex);
}
```
Bounded Buffer with Semaphores

n, the number of buffers, semaphores e, the number of empty buffers, f, the number of full buffers, and b, mutex

\[ e = n; f = 0; b = 1; \]

**Producer**

```java
while(true) {
    produce next record
    P(e); P(b);
    add to buffer
    V(b); V(f);
}
```

**Consumer**

```java
while(true) {
    P(f); P(b);
    remove from buffer
    V(b); V(e);
    process record
}
```

Semaphores are still low-level

- Semaphores can allow implementation of deadlock
  - Process one
    - P(s);
    - P(q);
    - critical section
      - P(s);
      - V(s);
    - V(q);
  - Process two
    - P(q);
    - P(s);
    - critical section
      - P(s);
      - V(s);
- Implementation might permit starvation (no guarantees)
- Consequently, even higher-level mechanisms have been developed (to be considered later)
Semaphore Implementation with Hardware atomic actions

- Implementation using test-and-set
- First implement binary semaphores
  - Value is either 0 or 1
- Use binary semaphores to implement general semaphores

Implementing binary semaphores

- Binary semaphores Pb(sb) and Vb(sb)
  - sb == false means we can pass
  - sb == true means we must wait
- Pb(sb): while (test-and-set(sb)) do ;
- Vb(sb): sb := false;
Implementing general semaphores

- binary semaphores: mutex and delay
  - P(s):
    
    ```
    Pb(mutex);
    s = s - 1;
    if(s < 0) then {Vb(mutex); Pb(delay);}
    Vb(mutex);
    ```
  - V(s):
    
    ```
    Pb(mutex);
    s = s + 1;
    if (s <= 0) then Vb(delay) else Vb(mutex);
    ```

Semaphore Implementation

- Disadvantage of using test-and-set is busy wait
- However, can implement P and V in more complex ways--for example, block process on P if the resource is busy with the subsequent V unblocking the next process to run (see text section 6.4.2).
Monitor: A Language Construct for Process Synchronization

Syntax of Monitor

```
type monitor-name = monitor
variable declarations
procedure entry P1 (...);
begin
   ...
end;

procedure entry Pn (...);
begin
   ...
end;

begin
   initialization code
end.
```

Semantic Rules

- Only one process can execute an entry procedure at any time.
- If a process calls an entry procedure while some process is inside the monitor, the caller is put on a waiting queue until the monitor is empty.

Notes about Monitors

- Monitors are a high-level data abstraction tool
- Based on abstract data types
  - For any distinct data type there should be a well-defined set of operations through which any instance of the data must be manipulated
  - Monitor is implemented as a collection of data (i.e., a resource) and a set of procedures that manipulate the resource
  - Access data only through the monitor procedures—outside procedures cannot access monitor’s variables
  - Similarly, monitor procedures only access monitor’s variables and formal parameters. Scoping rules followed within the monitor.
- Monitor is higher-level than P and V hence is safer and easier to use
Monitor Example

type atomic-write = monitor
var count: integer;
procedure entry aputs(s: string)
  begin
    count := count + 1;
    writeln(count, ': ', s);
  end;
begin
  count = 0;
end.

Monitor condition variables

- At most one procedure can be active in monitor at any time
  - Simplifies synchronization specification!
- What if a procedure has to wait for another procedure to act before continuing (for example: reading from an empty buffer)?
- Add the concept of condition variables
Conditional Variables in Monitor

Syntax:

Declaration: var x : condition

Use: x.wait and x.signal

Semantics:

A process invoking x.wait is suspended until another process invokes x.signal.

Question?
After x.signal, who should proceed?

Monitor Condition Variables

- P is executing and invokes x.signal; Q is waiting, having invoked x.wait in the past
- Should Q be permitted to resume execution immediately? What about P?
- Continuation options
  - Q remains suspended until P leaves the monitor or P performs a "wait"
  - P suspends and waits until Q either leaves the monitor or Q performs another "wait"
- First choice seems “fairer” since P already is executing
  - Condition Q waiting on may not still hold if P continues executing
- Second choice permits P to invoke multiple signals
- Both have been advocated in practice
Monitor Condition Variables

- Further details
  - if several processes are waiting on a condition, which is resumed?
    - This is a scheduling decision
  - if no process is waiting on a condition, what is the effect of an x.signal?
    - x.signal becomes a nop

Monitor Example

```plaintext
type bounded buffer = monitor
var buffer: array [0..n-1] of item;
  counter, in, out: integer;
  notempty, notfull: condition;
procedure entry add(item);
begin
  if (counter == n) then
    notfull.wait;
  buffer[in] := item;
  in := (in + 1) mod n;
  counter := counter + 1;
  notempty.signal;
end;

procedure entry remove(var item);
begin
  if (counter == 0) then
    notempty.wait;
  item := buffer[out];
  out := (out + 1) mod n;
  counter := counter - 1;
  notfull.signal;
end;
begin
  counter := 0; in := 0; out := 0;
end.
```
Dining Philosophers Problems

- n philosophers spend their lives thinking and eating.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks and eat.
- A philosopher may pick up one chopstick at a time. He needs two to eat. When he finishes, he releases the two chopsticks and starts thinking again.

Dining Philosophers Semaphore Solution

- Possible solution: each chopstick is a semaphore
  - \texttt{var} chopstick: \texttt{array [0..4] of semaphore};
  - \texttt{while(true)} {
    P(chopstick[i]);
    P(chopstick[(i+1) \mod 5]);
    \quad \textit{eat} \ldots
    V(chopstick[i]);
    V(chopstick[(i+1) \mod 5]);
    \quad \textit{think} \ldots
  }
  - possibility of deadlock (e.g., each philosopher grabs left chopstick simultaneously)
Dining Philosophers
Semaphore Solution

- So, simple solution is not adequate. Try some more complex solutions:
  - Pick up left, see if right available. If not, release left.
  - Odd philosophers pick up left first; even ones pick up right first
  - Philosopher picks up chopsticks only if both are available (an additional critical section)
    - Solution hint: add notion of state (HUNGRY, THINKING, EATING). Check state of neighbors.

- Goals
  - no starvation
  - maximal concurrency (n-1 philosophers can eat at the same time)

```
semaphores: mutex (initially == 1) for fork acquisition and return
phil(N), one per philosopher, to indicate that the philosopher is blocked awaiting fork release (initially == 0)

get_forks(i) {
  P(mutex);
  state[i] := HUNGRY;
  test(i); /* to be defined---test that forks are available and acquire if so */
  V(mutex);
  P(phil[i]); /* block if forks not available---see test() */
}

dining Philosophers
Semaphore Solution

put_forks(i) {
  P(mutex);
  state[i] := THINKING;
  test((i-1) mod N);
  test((i+1) mod N);
  V(mutex);
}

test(i) {
  if((state[i] == HUNGRY) &&
     (state[(i-1) mod N] != EATING) &&
     (state[(i+1) mod N] != EATING) &&
     
  )
    state[i] := EATING;
    V(phil[i]);
}
```
Dining Philosophers
Monitor Solution

declare dining-philosophers = monitor
var state: array[0..4] of (thinking, hungry, eating);
self: array[0..4] of condition;

procedure entry test(k: 0..4);
begin
  if state[k+4 mod 5] <> eating and
  state[k] = hungry and
  state[k+1 mod 5] <> eating
  then begin
    state[k] := eating;
    self[k].signal;
  end;
end;

procedure entry pickup(i: 0..4);
begin
  state[i] := hungry;
  test(i);
  if state[i] <> eating then
    self[i].wait;
end;

procedure entry putdown(i: 0..4);
begin
  state[i] := thinking;
  test(i+4 mod 5);
  test(i+1 mod 5);
end;

begin
  for i := 0 to 4
  do state[i] := thinking;
end.

Process Philosopher 1
begin
  repeat
    pickup(i);
    eating;
    putdown(i);
    thinking
  until false;
end.
Classical Problems
Readers/Writers (summary only)

- Two categories of processes accessing a data object
  - Readers (examine object)
  - Writers (modify object)
- Multiple readers can access object simultaneously without conflict
- Writer must have exclusive access to object, otherwise inconsistencies may arise
  - two writers modifying object simultaneously
  - reader getting inconsistent information because of access during write

Readers/Writers Summary

- Writers have exclusive access to object
- Reader behavior depends on definition of problem chosen
  - First readers-writers problem: no reader is kept waiting unless a writer has already obtained permission to use the shared object (i.e., readers don’t have to wait merely because a writer is waiting).
  - Second readers-writers problem: writer performs write as soon as possible once ready (i.e., no new readers can enter once a writer is waiting).
- See text figures 6.12 and 6.13 for one solution (pp. 183 and 184).