Chapter 6 : Process Synchronization

- Background
- Critical sections
- Hardware for synchronization
- Semaphores
- Classical synchronization problems
  - » Bounded-buffer
  - » Readers & writers
  - » Dining philosophers
- Critical regions
- Monitors

Background

- Multiple processes running at the same time may interleave their accesses to shared variables
  - » Processes can be interrupted anywhere
  - » Consistency must be maintained regardless of where switches occur
- Multiple processes need to synchronize amongst themselves to ensure
  - » Consistency of shared variables
  - » Orderly execution of code in different processes (A must execute before B, etc.)

Example: Bounded Buffer Problem

Shared variables

```
const int n;
typedef … Item;
Item buffer[n];
int in = 0, out = 0,
    counter = 0;
```

Atomic statements:

```
Counter += 1;
Counter -= 1;
```

Producer

```
Item pitm;
while (1) {
  …
  produce an item into pitm
  …
  while (counter == n)
    ;
  buffer[in] = pitm;
  in = (in+1) % n;
  counter += 1;
}
```

Consumer

```
Item citm;
while (1) {
  while (counter == 0)
    ;
  citm = buffer[out];
  out = (out+1) % n;
  counter -= 1;
  …
  consume the item in citm
  …
}
```


So Why Doesn't It Work?

- Modifying a variable has two parts
  - » Computing the new value for the variable
  - » Storing the new value into the variable
- Example: `counter = counter + 1`
  - » First, calculate `counter+1`
  - » Next, store the new value into `counter`
- Problem: two processes may modify the variable

Producer

```
…
A1 LOAD  r2,count
A2 ADD   r2,r2,#1
A3 STORE r2,count
…
```

Consumer

```
…
B1 LOAD  r3,count
B2 SUB   r3,r3,#1
B3 STORE r3,count
…
```

Solution: Critical Sections

- *n* processes competing to use some shared data
- Each process has a critical section in which the shared data is accessed
- At most one process may be in the critical section at any time
  - » No other process may execute in its critical section when one process is already there
  - » Other processes may need to wait
- General structure of a process:

```
while (1) {
  enter section
    critical section
  exit section
  rest of process
}
```


Critical Section Problem: Requirements

- Mutual Exclusion
  - » If process P_i is executing in its critical section, 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 want to enter their own critical sections, then the selection of the next process to enter the critical section can't be postponed indefinitely.
- Bounded Waiting
  - » A bound must exist on how many processes are allowed to enter their critical sections before a waiting process is allowed to enter its own critical section.
    - – Processes execute at non-zero speed
    - – No assumption about *relative* speed of processes

Solving the Critical Section Problem

- Only two processes, P0 and P1
- General structure of P_i (and other process P_j)

```
while (1) {
  enter critical section
    critical section
  exit critical section
  remainder of code
}
```

- Processes share variables to synchronize their actions

Critical Sections: First Try

- Satisfies mutual exclusion, but progress not guaranteed
  - » Problem: what if P_i wants to go twice in a row?
  - » P_i must wait for P_j to reset the `turn` variable

Shared variables

```
// turn == i means Pi can
// enter its critical
// section.
int turn = 0;
```

Process P_i

```
int i;  // my process ID
int j;  // other process ID
while (1) {
  while (turn != i)
    ;
  // critical section
  turn = j;
  // remainder of code
}
```

Critical Sections: Second Try

- Satisfies mutual exclusion, but not bounded waiting
  - Problem: P_i can exit the critical section and reenter it without allowing P_j to enter the critical section
  - This occurs if P_j is suspended in the middle of the waiting loop while P_i executes an entire loop

Shared variables

```
// flag[i] == 1 means Pi
// can enter its
// critical section.
int flag[2] = {0,0};
```

Process P_i

```
int i;  // my process ID
int j;  // other process ID
while (1) {
  flag[i] = 1;
  while (flag[j] == 1)
    ;
  // critical section
  flag[i] = 0;
  // remainder of code
}
```


Critical Sections: Third Try

- Combine first and second tries
- Satisfies all three requirements, solving the critical sections problem for two processes
  - P_i gives P_j a chance to enter before it does so itself

Shared variables

```
// flag[i] means that
// Pi wants to be in the
// critical section
int flag[2] = {0,0};
// turn==i means that
// Pi is allowed to
// enter c.s. if it wants
// to do so
int turn = 0;
```

Process P_i

```
int i;  // my process ID
int j;  // other process ID
while (1) {
  flag[i] = 1;
  turn = j;
  while (flag[j] && turn == j)
    ;
  // critical section
  flag[i] = 0;
  // remainder of code
}
```

What About More Than Two Processes?

- Critical section for *n* processes (*n*>=2)
- Use the bakery algorithm
  - » Each process gets a number before entering its critical section
  - » Holder of the smallest number enters the critical section
  - » Ties broken by allowing process with lowest process ID to go first (P_i goes before P_j if i<j)
  - » Numbers assigned in increasing order (such as 1,1,2,3,4,5,5,5,…)
  - » Each process receives a number that is strictly greater than the last number it received (so no process gets the same number twice)

Bakery Algorithm

- Notation used
  - » <<< is lexicographical order on (ticket#, process ID)
  - » (a,b) <<< (c,d) if (a<c) or ((a==c) and (b<d))
  - » Max($a_0, a_1, \ldots, a_{n-1}$) is a number k such that k>=a_i for all I
- Shared data
  - » `choosing` initialized to `0`
  - » `number` initialized to `0`

```
int n; // # of processes
int choosing[n];
int number[n];
```

Bakery Algorithm: Code

```
while (1) {  // i is the number of the current process
  choosing[i] = 1;
  number[i] = max(number[0],number[1],…,number[n-1]) + 1;
  choosing[i] = 0;
  for (j = 0; j < n; j++) {
    while (choosing[j])  // wait while j is choosing a
      ;                  // number
    // Wait while j wants to enter and has a better number
    // than we do.  In case of a tie, allow j to go if
    // its process ID is lower than ours
    while ((number[j] != 0) &&
           ((number[j] < number[i]) ||
            ((number[j] == number[i]) && (j < i))))
       ;
  }
  // critical section
  number[i] = 0;
  // rest of code
}
```


Hardware for Synchronization

- Prior methods work, but…
  - » May be somewhat complex
  - » Require <u>busy waiting</u>: process spins in a loop waiting for something to happen, wasting CPU time
- Solution: use hardware
- Several hardware methods
  - » Test & set: test a variable and set it in one instruction
  - » Atomic swap: switch register & memory in one instruction
  - » Turn off interrupts: process won't be switched out unless it asks to be suspended

Mutual Exclusion Using Hardware

- Single shared variable `lock`
- Still requires busy waiting, but code is much simpler
- Two versions
  - » Test and set
  - » Swap
- Works for any number of processes
- Possible problem with requirements
  - » Non-concurrent code can lead to unbounded waiting

```
int lock = 0;
```

Code for process P_i

```
while (1) {
  while (TestAndSet(lock))
    ;
  // critical section
  lock = 0;
  // remainder of code
}
```

Code for process P_i

```
while (1) {
  while (Swap(lock,1) == 1)
    ;
  // critical section
  lock = 0;
  // remainder of code
}
```


Eliminating Busy Waiting

- Problem: previous solutions waste CPU time
  - » Both hardware and software solutions require spin locks
  - » Allow processes to sleep while they wait to execute their critical sections
- Solution: use semaphores
  - » Synchronization mechanism that doesn't require busy waiting
- Implementation
  - » Semaphore S accessed by two atomic operations
    - – Wait(S): while (S<=0) {}; S-= 1;
    - – Signal(S): S+=1;
  - » Wait() is another name for P()
  - » Signal() is another name for V()
  - » Modify implementation to eliminate busy wait from Wait()

Critical Sections Using Semaphores

- Define a class called `Semaphore`
  - » Class allows more complex implementations for semaphores
  - » Details hidden from processes
- Code for individual process is simple

Shared variables

```
Semaphore mutex;
```

Code for process P_i

```
while (1) {
  wait(mutex);
  // critical section
  signal(mutex);
  // remainder of code
}
```


Implementing Semaphores with Blocking

- Assume two operations:
  - » Block(): suspends current process
  - » Wakeup(P): allows process P to resume execution
- Semaphore is a class
  - » Track value of semaphore
  - » Keep a list of processes waiting for the semaphore
- Operations still atomic

```
class Semaphore {
  int value;
  ProcessList pl;
  void Wait ();
  void Signal ();
};
```

Semaphore code

```
Semaphore::Wait ()
{
  value -= 1;
  if (value < 0) {
    // add this process to pl
    Block ();
  }
}
Semaphore::Signal () {
Process P;
  value += 1;
  if (value <= 0) {
    // remove a process P
    // from pl
    Wakeup (P);
  }
}
```

Semaphores for General Synchronization

- We want to execute B in P_1 only after A executes in P_0
- Use a semaphore initialized to 0
- Use Signal() to notify P_1 at the appropriate time

Shared variables
```
// flag initialized to 0
Semaphore flag;
```

Process P_0
```
   .
   .
   .
// Execute code for A
flag.Signal ();
```

Process P_1
```
   .
   .
   .
flag.Wait ();
// Execute code for B
```


Types of Semaphores

- Two different types of semaphores
  - » Counting semaphores
  - » Binary semaphores
- Counting semaphore
  - » Value can range over an unrestricted range
- Binary semaphore
  - » Only two values possible
    - – 1 means the semaphore is available
    - – 0 means a process has acquired the semaphore
  - » May be simpler to implement
- Possible to implement one type using the other

Using Binary Semaphores

```
class Semaphore {
  int count;
  BinSem S1(1),S2(0),S3(1);
  void Wait ();
  void Signal();
}
```

```
Semaphore::Signal ()
{
  S1.Wait();
  count += 1;
  if (count <= 0) {
    S2.Signal();
  }
  S1.Signal();
}
```

```
Semaphore::Wait()
{
  S3.Wait();
  S1.Wait();
  count -= 1;
  if (count < 0) {
    S1.Signal ();
    S2.Wait ();
  } else {
    S1.Signal ();
  }
  S3.Signal ();
}
```


Deadlock and Starvation

- Deadlock: two or more processes are waiting indefinitely for an event that can only by caused by a waiting process
  - » P_0 gets A, needs B
  - » P_1 gets B, needs A
  - » Each process waiting for the other to signal
- Starvation: indefinite blocking
  - » Process is never removed from the semaphore queue in which its suspended
  - » May be caused by ordering in queues (priority)

Shared variables

```
Semaphore A(1),B(1);
```

Process P_0

```
A.Wait();
B.Wait();
   .
   .
   .
B.Signal();
A.Signal();
```

Process P_1

```
B.Wait();
A.Wait();
   .
   .
   .
A.Signal();
B.Signal();
```

Classical Synchronization Problems

- Bounded Buffer
  - » Multiple producers and consumers
  - » Synchronize access to shared buffer
- Readers & Writers
  - » Many processes that may read and/or write
  - » Only one writer allowed at any time
  - » Many readers allowed, but not while a process is writing
- Dining Philosophers
  - » Resource allocation problem
  - » *N* processes and limited resources to perform sequence of tasks
- Goal: use semaphores to implement solutions to these problems

Bounded Buffer Problem

- Goal: implement producer-consumer without busy waiting

```
const int n;
Semaphore empty(n),full(0),mutex(1);
Item buffer[n];
```

Producer

```
int in = 0;
Item pitem;
While (1) {
  // produce an item
  // into pitem
  empty.Wait();
  mutex.Wait();
  buffer[in] = pitem;
  in = (in+1) % n;
  mutex.Signal();
  full.Signal();
}
```

Consumer

```
int out = 0;
Item citem;
While (1) {
  full.Wait();
  mutex.Wait();
  citem = buffer[out];
  out = (out+1) % n;
  mutex.Signal();
  empty.Signal();
  // consume item from
  // citem
}
```

Readers-Writers Problem

Shared variables

```
int nreaders;
Semaphore mutex(1), writing(1);
```

Reader process

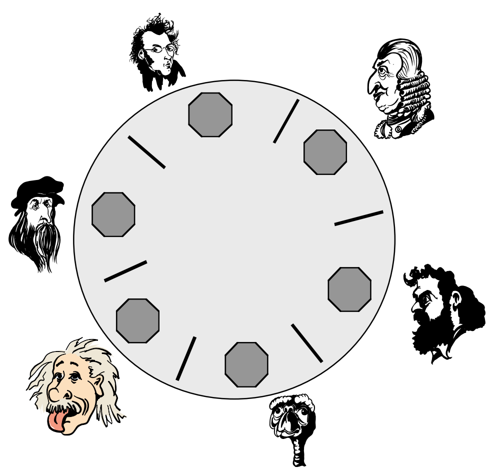
```
…
mutex.Wait();
nreaders += 1;
if (nreaders == 1) // wait if
  writing.Wait();  // 1st reader
mutex.Signal();
// Read some stuff
mutex.Wait();
nreaders -= 1;
if (nreaders == 0)  // signal if
  writing.Signal(); // last reader
mutex.Signal();
…
```

Writer process

```
…
writing.Wait();
// Write some stuff
writing.Signal();
…
```


Dining Philosophers

- *N* philosophers around a table
  - » All are hungry
  - » All like to think
- *N* chopsticks available
  - » 1 between each pair of philosophers
- Philosophers need two chopsticks to eat
- Philosophers alternate between eating and thinking
- Goal: coordinate use of chopsticks



Dining Philosophers: Solution 1

- Use a semaphore for each chopstick
- A hungry philosopher
  - » Gets the chopstick to his right
  - » Gets the chopstick to his left
  - » Eats
  - » Puts down the chopsticks
- Potential problems?
  - » Deadlock
  - » Fairness

Shared variables

```
const int n;
// initialize to 1
Semaphore chopstick[n];
```

Code for philosopher *i*

```
while(1) {
  chopstick[i].Wait();
  chopstick[(i+1)%n].Wait();
  // eat
  chopstick[i].Signal();
  chopstick[(i+1)%n].Signal();
  // think
}
```


Dining Philosophers: Solution 2

- Use a semaphore for each chopstick
- A hungry philosopher
  - » Gets lower, then higher numbered chopstick
  - » Eats
  - » Puts down the chopsticks
- Potential problems?
  - » Deadlock
  - » Fairness

Shared variables

```
const int n;
// initialize to 1
Semaphore chopstick[n];
```

Code for philosopher *i*

```
int i1,i1;
while(1) {
  if (i != (n-1)) {
    i1 = i;
    i2 = i+1;
  } else {
    i1 = 0;
    i2 = n-1;
  }
  chopstick[i1].Wait();
  chopstick[i2].Wait();
  // eat
  chopstick[i1].Signal();
  chopstick[i2].Signal();
  // think
}
```

Different Synchronization Mechanisms

- Semaphores are good, but...
  - » Prone to programming errors
    - – Reverse order of operations
    - – Forget to Signal() after Wait()
  - » Require effort from programmers to get it right
  - » Don't provide high-level view of structures
- Consider alternate synchronization mechanisms
  - » Critical regions
  - » Monitors
  - » Locks & condition variables

Critical Regions

- More general solution to accessing shared variables
  - » Shared variables accessed within regions
  - » Regions referring to the same shared variable exclude each other, limiting access to one process at a time
  - » Different processes can access different regions that don't use the same variables simultaneously
- Increased flexibility
  - » Allows more simultaneous execution
  - » Enforces mutual exclusion - harder to make programming errors
- Solution provided in some languages
  - » Not provided in standard C/C++
  - » Can be emulated using semaphores

Critical Regions: Details

- Region usage:
  `region` *`r`* `when` *`cond`*
  `{`*`actions`*`}`
- Only one process can be in a region labeled *r*
  - » Multiple labels allow different sets of mutual exclusion regions
- A process can enter region only when condition *cond* is true
  - » Condition evaluated in mutual exclusion as well
- Critical regions can be implemented using semaphores

Monitors

- A monitor is another kind of high-level synchronization primitive
  - » One monitor has multiple entry points
  - » Only one process may be in the monitor at any time
  - » Enforces mutual exclusion - less chance for programming errors
- Monitors provided by high-level language
  - » Variables belonging to monitor are protected from simultaneous access
  - » Procedures in monitor are guaranteed to have mutual exclusion
- Monitor implementation
  - » Language / compiler handles implementation
  - » Can be implemented using semaphores

Monitor Usage

```
monitor mon {
  int foo;
  int bar;
  double arr[100];
  void proc1(…) {
  }
  void proc2(…) {
  }
  void mon() {  // initialization code
  }
};
```

- This looks like C++ code, but it's not supported by C++
- Provides the following features:
  - » Variables foo, bar, and arr are accessible only by proc1 & proc2
  - » Only one process can be executing in either proc1 or proc2 at any time

Condition Variables in Monitors

- Problem: how can a process wait inside a monitor?
  - » Can't simply sleep: there's no way for anyone else to enter
  - » Solution: use a condition variable
- Condition variables support two operations
  - » Wait(): suspend this process until signaled
  - » Signal(): wake up exactly one process waiting on this condition variable
    - – If no process is waiting, signal has no effect
    - – Signals on condition variables aren't "saved up"
- Condition variables are only usable within monitors
  - » Process must be in monitor to signal on a condition variable
  - » Question: which process gets the monitor after Signal()?

Monitor Semantics

- Problem: P signals on condition variable X, waking Q
  - » Both can't be active in the monitor at the same time
  - » Which one continues first?
- Mesa semantics
  - » Signaling process (P) continues first
  - » Q resumes when P leaves the monitor
  - » Seems more logical: why suspend P when it signals?
- Hoare semantics
  - » Awakened process (Q) continues first
  - » P resumes when Q leaves the monitor
  - » May be better: condition that Q wanted may no longer hold when P leaves the monitor
- For project, use Mesa semantics

Locks & Condition Variables

- Monitors require native language support
- Provide monitor support using special data types and procedures
  - » Locks (Acquire(), Release())
  - » Condition variables (Wait(), Signal())
- Lock usage
  - » Acquiring a lock == entering a monitor
  - » Releasing a lock == leaving a monitor
- Condition variable usage
  - » Each condition variable is associated with exactly one lock
  - » Lock must be held to use condition variable
  - » Waiting on a condition variable releases the lock implicitly
  - » Returning from Wait() on a condition variable reacquires the lock

Dining Philosophers with Locks

Shared variables

```
const int n;
// initialize to THINK
int state[n];
Lock mutex;
// use mutex for self
Condition self[n];
```

```
void test(int k)
{
  if ((state[(k+n-1)%n)]!=EAT) &&
      (state[k]==HUNGRY) &&
      (state[(k+1)%n]!=EAT)) {
    state[k] = EAT;
    self[k].Signal();
  }
}
```

```
Code for philosopher j
while (1) {
  // pickup chopstick
  mutex.Acquire();
  state[j] = HUNGRY;
  test(j);
  if (state[j] != EAT)
    self.Wait();
  mutex.Release();
  // eat
  mutex.Acquire();
  state[j] = THINK;
  test((j+1)%n); // next
  test((j+n-1)%n); // prev
  mutex.Release();
  // think
}
```


Implementing Locks with Semaphores

```
class Lock {
  Semaphore mutex(1);
  Semaphore next(1);
  int nextCount = 0;
};
```

```
Lock::Acquire()
{
  mutex.Wait();
}
```

```
Lock::Release()
{
  if (nextCount > 0)
    next.Signal();
  else
    mutex.Signal();
}
```

- Use `mutex` to ensure exclusion within the lock bounds
- Use `next` to give lock to processes with a higher priority (why?)
- `nextCount` indicates whether there are any higher priority waiters

Implementing Condition Variables

```
class Condition {
  Lock *lock;
  Semaphore condSem(0);
  int semCount = 0;
};
```

```
Condition::Wait ()
{
  semCount += 1;
  if (lock->nextCount > 0)
    lock->next.Signal();
  else
    lock->mutex.Signal();
  condSem.Wait ();
  semCount -= 1;
}
```

```
Condition::Signal ()
{
  if (semCount > 0) {
    lock->nextCount += 1;
    condSem.Signal ();
    lock->next.Wait ();
    lock->nextCount -= 1;
  }
}
```

- Are these Hoare or Mesa semantics?
- Can there be multiple condition variables for a single Lock?