Benjamin Fenster
CS-201 (Arslan)
14 November 2005

Notes – Process Synchronization

I. Introduction
   a. If processes share no resources then you can schedule them by any policy that meets the system's needs.
   b. When processes share resources, concurrent access to data may cause inconsistency.
   c. Example
      i. Have a bounded buffer; one producer, one consumer.
      ii. Producer waits 'till the buffer is not full, inserts items at [in % BUFFER_SIZE]
      iii. Consumer waits until the buffer isn't empty, removes from [out mod BUFFER_SIZE]
      iv. 'counter' is incremented /decremented as items are inserted / removed
         1. In machine language counter++; may be
            register1 = counter
            register1 = register1 + 1
            counter = register
         2. counter--; may be:
            register2 = counter
            register2 = register2 – 1
            counter = register
      v. Now there's a problem! What if the process is interrupted in the middle of those three steps?
         1. May have counter = 4, register1 becomes 5
         2. Now switch to the consumer process.
         3. Counter is still 4, so register2 becomes 3.
         4. Store 3 back in counter.
         5. Now return to the producer and store 5 back in counter.
   d. We need a way to describe which parts of a process must be synchronized.
   e. The example is called a race condition: several processes are accessing the same memory, so the final value depends on whoever gets there first.

II. Critical Section
   a. Assume there are n processes; each has a part that accesses shared data
   b. That part of the code is called the critical section
   c. Constraints on an Ideal Solution
      i. Mutual Exclusion
         1. If a process is executing in its critical section, no other process can execute in its critical section.
         2. Thus only one process can access the shared data
      ii. Progress
         1. If no process is in its critical section, but one wants to start, it should be allowed.
         2. Execution cannot be indefinitely postponed.
      iii. Bounded Waiting
         1. When a process asks to execute in its critical section, there must be a bound for the number of times other processes will execute in their critical sections.
         2. There is no assumption about the relative speed of processes – one process may take much longer or shorter.

III. Candidate Solutions
   a. Consider the case with only two processes: p_1, p_2
      i. do { entry section; critical section; exit section; remainder; } while (true);
      ii. p_1 do {

```
wait (turn != 1);
critical section
turn = 2
```

```
            remainder section
      } while (true);
```

   iii. (With p_2 symmetrical)
   iv. Mutual exclusion is satisfied.
   v. Progress is not. p_1 may be postponed indefinitely until p_2 finishes.
   vi. This candidate solution *fails*.

b. Another candidate
   i.

```
do {  flag[i] = true;
      while (flag[j]);
      critical section
      flag[i] = false;
      remainder section
} while (true);
```

   ii. Could have a deadlock if both flag[1], flag[2] = true.
   iii. Mutual exclusion is satisfied.
   iv. Progress is not (could deadlock).
   v. Solution *fails*

c. Another candidate
   i.

```
do {  flag[i] = true;
      turn = j;
      while (flag[j] && turn == j);
      critical section
      flag[i] = false;
      remainder section
} while (true);
```

   ii. Mutual exclusion is still satisfied
   iii. Avoids deadlocks now sine 'turn' can be only one value
   iv. Proving that this complies requires considering all possible scenarios (stating loop invariants, et cetera
   v. We won't formally prove it, but get the idea
   vi. For multiple (more than two) processes, Baker's Algorithm
      1. Wait as long as another process is taking a ticket or if any process has a lower ticket number
      2. (Simulating a bakery or deli line where everybody has a number and the next person is served)
      3.

```
do {  choosing[i] = true;
      number[i] = max(number[0], ...,
                        number[n-1]) + 1;
      choosing[i] = false;
      for (j = 0; j < n; j++)
            while (choosing[j]);
            while (number[j] != 0
                  && (number[j] < number[i]
                  || (number[j]==number[i] && i>=j)
            );
      }
      critical section
      number[i] = 0;
      remainder section;
} while (true);
```

d. NB: In reality these algorithms are not implemented in code but with hardware support.

e. Another Solution
   i. Uses hardware support
   ii.

```
boolean TestAndSet (boolean& target) {
      boolean rv = target;
      target = true;
      return rv;
}
```

   iii.

```
Pi: do while (TestAndSet(lock));
      critical section;
      lock = false;
   remainder section;
}
```

   iv. Hardware guarantees that the entire TestAndSet() function will be evaluated atomically.
   v. TestAndSet() means, “I want to use the critical section, so set the lock.” If there was already a lock, returns true, meaning “wait.”

f. Another Solution
   i. Also uses an atomic function
   ii.

```
void swap (boolean& a, boolean& b) {
      boolean temp = a; a = b; b = temp;
}
```

   iii.

```
Pi: do key = true;
      while (key == true) swap (lock, key);
      critical section;
      lock = false;
}
```

   iv. If the lock was true, key stays true (lock stays true).
   v. If the lock was false, key = false, lock = true

g. The Rub
   i. These are all very simple solutions, but processes spend a lot of CPU time just waiting.
   ii. Since the process that’s executing its critical section knows when it finishes, it could notify that it’s done.

IV. Semaphores

a. A new data structure
b. Semaphore
   i. s: an integer value
   ii. wait(s): while (s <= 0) do no-op; s--;
   iii. signal(s): s++
   iv. Both wait() and signal() are atomic.
c. How to Use: do { wait(mutex); critical section; signal(mutex); remainder section; }
d. Operations would actually be implemented differently:
   i. wait(s): s.value--; if (value < 0) { /* add this value to the waiting list */ block; }
   ii. block; is a syscall that blocks execution
   iii. signal(s): s.value++; if (value <= 0) { /* remove p from waiting list */ wakeup p; }
e. Example
   i. We have many producers and consumers using a bounded buffer (of size n)
   ii. Have semaphores empty = n, mutex = 1, full = 0
   iii. Producer p:

```
do {
      Produce an item in ‘nextp’
      wait(empty);
```

```
            wait(mutex);
            add nextp to the buffer
            signal(mutex)

            signal(full);
      } while (true);
```

iv. Consumer c:

```
      do {
            wait(full);

            wait(mutex);
            remove next from the buffer
            signal(mutex)

            signal(empty);

            Consume nextc
      } while (true);
```

v. empty starts out at n, so it'll only hit zero when there are zero empty slots.
vi. full starts out at 0 and can increment up to n.

f. Two Kinds of Sempahores:
   i. Counting: What we just used
   ii. Binary: Can only be 0 or 1 (duh)
      1. Can implement counting semaphores using binary semaphores by having a binary semaphore to control access to an ordinary counting variable.

V. Deadlocks

a. Introduction
   i. P_0 says: wait(s); wait(q); … signal(s); signal(q);
   ii. P_1 says: wait(q); wait(s); … signal(q); signal(s);
   iii. Ve haff a deadlock, kiptin!
   iv. Definition of Deadlock: Two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

b. Dining Philosophers Problem
   i. Five philosophers sit at a circular table with one chopstick between each pair.
   ii. To eat, a philosopher needs both chopsticks (i.e. one from each side)
   iii. How can we synchronize these processes?
   iv. P_i:

```
      do {
            wait(chopstick[i]); wait(chopstick[(i+1) % 5]);
            eat;
            signal(chopstick[i]); signal(chopstick[(i+1) % 5]);
            think; philosophize;
      } while (true);
```

   v. Again, a deadlock is possible. Each philosopher gets one chopstick and waits for the second (which will be held forever by her neighbor).
   vi. We want a deadlock-free solution.
   vii. Perhaps odd-numbered philosophers might reach for the left chopstick first, then the right (while even-numbered philosophers would do the opposite). Presto!

c. Monitors
   i. Programming Language supported construct
   ii. Skeleton:

```
      monitor monitor-name{
            shared variable declarations
            procedure body Pi (…) { }
            other procedure bodies
```

```
        void init() { }
    }
```

iii. Processes that want to execute inside the monitor are queued

iv. Conditions:

1. Two operations: wait() and signal()
2. wait: Process executing x.wait() enters a queue waiting for condition x and is suspended until someone does x.signal().

v. Example: Producer / consumer

1. Defined inside a ProducerConsumer monitor:

```
condition full, empty;
integer count;
procedure insert(item : integer) begin
      if count = N then full.wait()
      insert_item(item) // defined somewhere
      count = count + 1
      if count = 1 then empty.signal()
end
function remove() : integer begin
      if count = 0 then empty.wait()
      remove = remove_item()   // defined somewhere
      count = count – 1
      if count = N – 1 then full.signal()
end
count = 0
```

2. Producer:

```
begin
      while true do begin
            item = produce_item
            ProducerConsumer.insert(item)
      end
end
```

3. Consumer:

```
begin
      while true do begin
            item = ProducerConsumer.remove
            consume_item(item)
      end
end
```

vi. The Programming language guarantees synchronization. Can be implemented using semaphores (created automatically by the compiler).

VI. Sleeping Barber Problem

a. Another synchronization problem

b. Problem:

i. A waiting room has n chairs

ii. The barber takes a nap when there are no customers

iii. A customer leaves when there are no chairs

iv. A customer wakes up the barber if he is asleep.

c. Using semaphores:

```
semaphore customers = 0, barber = 0, mutex = 1
int waiting = 0
```

d. Barber:

```
while (true) {
      wait(customer);   // inherently means sleeping if there are
no customers
      wait(mutex);
      waiting--;
```

```
      signal(barber)
      signal(mutex);
      cut_hair();
}
```

e. Customer

```
wait(mutex);
if (waiting < n) {
      waiting++;
      signal(customer);
      signal(mutex);
      wait(barber);
      get_haircut();
} else {
      signal(mutex);
}
```

VII. Critical Regions
   a. Another high-level synchronization construct similar to monitors
   b. Shared variables declared as v : shared τ
   c. region v when B do S
      i. B is some boolean expression
      ii. S is some statement
      iii. S is executed only when B is true, and while it's executing no other process can access v.
   d. Guaranteed by the underlying high-level language (will translate to statements using semaphores).