CSC 539: Operating Systems Structure and Design

Spring 2006

Process synchronization

- critical section problem
- synchronization hardware
- semaphores
- high-level constructs: critical regions, monitors, synchronized
- synchronization in Solaris & Windows XP

1

Process synchronization

as we discussed earlier, processes can cooperate

- interaction can occur through shared data (e.g., threads) or shared files
- cooperating processes allow information sharing, computational speedup, modularity, and convenience
- synchronization is an issue even for uniprocessor systems

representative example: producer-consumer problem

two processes: producer and consumer

- producer generates data and places it in a bounded buffer
- consumer accesses and uses the data in the buffer
- the two processes must coordinate access of the shared the buffer
  keep count of unconsumed data, check to see if empty or full

2

Producer-consumer solution?

Producer

```
in = 0;
while (true) {
    PRODUCE nextP
    while (counter == BUFFER_SIZE) {}

    buffer[in] = nextP;
    in = (in + 1) % BUFFER_SIZE;

    counter++;
}
```

Consumer

```
out = 0;
while (true) {
    while (counter == 0) {}

    nextC = buffer[out]
    out = (out + 1) % BUFFER_SIZE;

    counter--;
    CONSUME nextC
}
```

shared counter keeps track of number of unconsumed data items

- assuming counter is initially 0, each process is correct in isolation

however, incrementing/decrementing the counter is NOT an atomic operation

`counter++;` (in machine code)

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--;` (in machine code)

```
register2 = counter
register2 = register2 - 1
counter = register2
```

potential problems?

3

Race conditions

if the producer and consumer attempt to update the counter simultaneously (e.g., one operation is preempted by the other), the assembly language statements may get interleaved

INTERLEAVING 1	INTERLEAVING 2
(p) register$_1$ = counter	(p) register$_1$ = counter
(p) register$_1$ = register$_1$ + 1	(p) register$_1$ = register$_1$ + 1
(c) register$_2$ = counter	(c) register$_2$ = counter
(c) register$_2$ = register$_2$ - 1	(c) register$_2$ = register$_2$ - 1
(c) counter = register$_2$	(p) counter = register$_1$
(p) counter = register$_1$	(c) counter = register$_2$

race condition: when multiple processes access & manipulate shared data, the final value of the data may depend on the interleaving order

- first interleaving yields counter+1, second yields counter-1 → neither is correct!
- to prevent race conditions, concurrent processes must be *synchronized*

4

Concrete Java example

Java provides a Thread class for creating and running "concurrent" threads

- can define a parent class to encapsulate shared data (e.g., array & counter)
- derived classes define the different behaviors of the Producer and Consumer

```
public class ProducerConsumer extends Thread
{
  protected static final int REPETITIONS = 10;
  protected static final int BUFFER_SIZE = 4;

  protected static int[] nums;
  protected static int counter;

  public ProducerConsumer()
  {
    nums = new int[BUFFER_SIZE];
    counter = 0;
  }
}
```

```
public class Producer extends ProducerConsumer
{
  public void run()
  {
    int in = 0;
    for (int i = 1; i <= REPETITIONS; i++) {
      while (counter == nums.length) { }

      nums[in] = i;
      in = (in+1)%nums.length;

      System.out.println("Producer: " + i);
      try {
        sleep((int)(Math.random()*500));
      }
      catch (InterruptedException e) { }

      counter++;
    }
  }
}
```

```
public class Consumer extends ProducerConsumer
{
  public void run()
  {
    int out = 0, sum = 0;
    for (int i = 1; i <= REPETITIONS; i++) {
      while (counter == 0) { }

      sum += nums[out];
      out = (out+1)%nums.length;

      System.out.println("Consumer: " + sum);
      try {
        sleep((int)(Math.random()*500));
      }
      catch (InterruptedException e) { }

      counter--;
    }
  }
}
```

5

Java execution

consider the following test program that uses a Producer & Consumer to calculate the sum from 1 to 10

```
public class PCTest
{
  public static void main(String[] args)
  {
      Producer p = new Producer();
      Consumer c = new Consumer();

      p.start();
      c.start();
  }
}
```

```
Producer: 1
Producer: 2
Consumer: 1
Producer: 3
Consumer: 3
Producer: 4
Consumer: 6
Producer: 5
Consumer: 10
Producer: 6
Consumer: 15
Producer: 7
Consumer: 21
Producer: 8
Consumer: 28
Producer: 9
Producer: 10
Consumer: 36
Consumer: 45
Consumer: 55
```

```
Producer: 1
Producer: 2
Consumer: 1
Producer: 3
Consumer: 3
Producer: 4
Consumer: 6
Producer: 5
Producer: 6
Consumer: 10
Producer: 7
Consumer: 15
Producer: 8
Consumer: 21
Producer: 9
Consumer: 28
Producer: 10
Consumer: 36
Consumer: 45
Consumer: 55
```

note that the random delay in the code means that the order of execution can vary

the while loops makes sure the

- Producer can't write if the buffer is full
- Consumer can't access if buffer is empty

Java does not dictate how threads are to be scheduled

- up to JVM and/or the underlying operating system

6

Critical section problem

suppose have N processes competing to use some shared data

- each process has a code segment (critical section) in which the data is accessed
- must ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section

each process must request permission to enter its critical section (entry)
then notify other processes when done (exit)

```
while (true) {
  entry section
   critical section
  exit section
   remainder section
}
```

any solution to critical section problem must satisfy:

- *mutual exclusion:* only one process can be in its critical section at any given time
- *progress:* only processes outside remainder section can influence choice of next process, and decision can't be postponed indefinitely
- *bounded waiting:* there is a limit on number of times a process waiting for critical section can be passed over in favor of other processes

7

Algorithm 1

for simplicity, assume 2 processes (P_0 and P_1)

can use a common variable to decide whose turn it is

- initially, `turn = 0`
- if (`turn == i`), then P_i can enter its critical section

P_i:

```
while (true) {
  while (turn != i) {}
   critical section
  turn = (i+1)%2;
   remainder section
}
```

this solution ensures *mutual exclusion*

but does not ensure *progress*

- requires strict alternation of execution
- P_1 might wait while P_0 executes remainder section

problem: need to know who wants their critical section & be responsive

8

Concrete Java example

in our Java code

- can define a shared turn field, which alternates between the PRODUCER and the CONSUMER

```
public class ProducerConsumer extends Thread
{
    protected static final int PRODUCER=0, CONSUMER=1;
    protected static final int REPETITIONS = 10;
    protected static final int BUFFER_SIZE = 5;

    protected static int[] nums;
    protected static int counter;
    protected static int turn;

    public ProducerConsumer()
    {
        nums = new int[BUFFER_SIZE];
        counter = 0;
        turn = PRODUCER;
    }
}
```

```
public class Producer extends ProducerConsumer
{
  public void run()
  {
    int in = 0;
    for (int i = 1; i <= REPETITIONS; i++) {
      while (counter == nums.length) { }

      nums[in] = i;
      in = (in+1)%nums.length;

      System.out.println("Producer: " + i);
      try { … } catch ( … ) { }

      while (turn != PRODUCER) { }
      counter++;
      turn = CONSUMER;
    }
  }
}
```

```
public class Consumer extends ProducerConsumer
{
  public void run()
  {
    int out = 0, sum = 0;
    for (int i = 1; i <= REPETITIONS; i++) {
      while (counter == 0) { }

      sum += nums[out];
      out = (out+1)%nums.length;

      System.out.println("Consumer: " + sum);
      try { … } catch ( … ) { }

      while (turn != CONSUMER) { }
      counter--;
      turn = PRODUCER;
    }
  }
}
```

9

Java execution

using the modified Producer and Consumer (with turn field):

```
Producer: 1
Producer: 2
Consumer: 1
Producer: 3
Consumer: 3
Producer: 4
Consumer: 6
Producer: 5
Consumer: 10
Producer: 6
Consumer: 15
Producer: 7
Consumer: 21
Producer: 8
Consumer: 28
Producer: 9
Consumer: 36
Producer: 10
Consumer: 45
Consumer: 55
```

```
Producer: 1
Producer: 2
Consumer: 1
Producer: 3
Consumer: 3
Producer: 4
Consumer: 6
Producer: 5
Consumer: 10
Producer: 6
Consumer: 15
Producer: 7
Consumer: 21
Producer: 8
Consumer: 28
Producer: 9
Consumer: 36
Producer: 10
Consumer: 45
Consumer: 55
```

the Producer and Consumer alternate

why does the Producer go twice at the beginning?

10

Algorithm 2

instead of using a variable to keep track of turns, could use flags to keep track of the processing waiting on critical section

- initially, `flag[0] = false, flag[1] = false`
- if (`flag[i] == true`), then P_i wants to enter its critical section

P_i:

```
while (true) {
    flag[i] = true;
    while (flag[(i+1)%2]) {}

    critical section

    flag[i] = false;

    remainder section
}
```

this solution ensures *mutual exclusion*

unlike Algorithm 1, it doesn't require alternation

however, it still does not ensure *progress*

- sensitive to interleaving
  (P_0) `flag[0] = true;`
  (P_1) `flag[1] = true;`
  `. . .`

11

Algorithm 3: Peterson's solution

can combine the variable & flags from Algorithms 1 & 2 to get a solution

- flags ensure that if a process is the only one waiting, it will get critical section
- turn variable ensures that the other process will get a chance

P_i:

```
while (true) {
    flag[i] = true;
    turn = (i+1)%2;
    while (flag[(i+1)%2] &&
           turn == (i+1)%2) {}

    critical section

    flag[i] = false;

    remainder section
}
```

this solution ensures *mutual exclusion*

- critical section entered only if either
  `flag[(i+1)%2] == 0` → other process not ready
  OR
  `turn != (i+1)%2` → not other process' turn

it also ensures *progress* & *bounded waiting*

- since turn is either 0 or 1, one must get through
- if other process waiting, gets next turn

can be easily generalized to N processes

12

Concrete Java example

in our Java code

- can add an additional pair of boolean variables, initialized to false

```
public class ProducerConsumer extends Thread
{
    protected static final int PRODUCER=0, CONSUMER=1;
    protected static final int REPETITIONS = 10;
    protected static final int BUFFER_SIZE = 5;

    protected static int[] nums;
    protected static int counter;
    protected static boolean[] flag = {false, false};
    protected static int turn;

    public ProducerConsumer()
    {
        nums = new int[BUFFER_SIZE];
        counter = 0;
        turn = PRODUCER;
    }
}
```

```
public class Producer extends ProducerConsumer
{
  public void run()
  {
    int in = 0;
    for (int i = 1; i <= REPETITIONS; i++) {
      while (counter == nums.length) { }

      nums[in] = i;
      in = (in+1)%nums.length;

      System.out.println("Producer: " + i);
      try { … } catch ( … ) { }

      flag[PRODUCER] = true;
      turn = CONSUMER;
      while (flag[CONSUMER] && turn==CONSUMER){}
      counter++;
      flag[PRODUCER] = false;
    }
  }
}
```

```
public class Consumer extends ProducerConsumer
{
  public void run()
  {
    int out = 0, sum = 0;
    for (int i = 1; i <= REPETITIONS; i++) {
      while (counter == 0) { }

      sum += nums[out];
      out = (out+1)%nums.length;

      System.out.println("Consumer: " + sum);
      try { … } catch ( … ) { }

      flag[CONSUMER] = true;
      turn = PRODUCER;
      while (flag[PRODUCER] && turn==PRODUCER){}
      counter--;
      flag[CONSUMER] = false;
    }
  }
}
```

13

Java execution

using the modified Producer and Consumer (with flag & turn):

```
Producer: 1
Producer: 2
Consumer: 1
Producer: 3
Consumer: 3
Producer: 4
Producer: 5
Consumer: 6
Producer: 6
Consumer: 10
Consumer: 15
Producer: 7
Consumer: 21
Producer: 8
Consumer: 28
Producer: 9
Consumer: 36
Producer: 10
Consumer: 45
Consumer: 55
```

```
Producer: 1
Producer: 2
Consumer: 1
Producer: 3
Consumer: 3
Producer: 4
Consumer: 6
Producer: 5
Producer: 6
Consumer: 10
Consumer: 15
Producer: 7
Consumer: 21
Producer: 8
Producer: 9
Consumer: 28
Consumer: 36
Producer: 10
Consumer: 45
Consumer: 55
```

note that the order can vary

14

Synchronization hardware

hardware can always help solve software problems

disabling interrupts

- as process enters critical section, it disables the interrupt system
- can be used in non-preemptive systems
- doesn't work for multiprocessors WHY?

atomic CPU (machine language) instructions
e.g., test-and-set instruction

```
bool TestAndSet(bool & target)
// sets target to true,
// but returns initial value
```

P_i:

```
while (true) {
  while (TestAndSet(lock)) {}

  critical section

  lock = false;

  remainder section
}
```

using test-and-set, can protect critical section

- this simple example fails bounded wait
- see text for complex but complete solution

15

Concrete Java example

in Java, the `synchronized` keyword can be used to define atomic operations

```
public class ProducerConsumer extends Thread
{
    protected static final int REPETITIONS = 10;
    protected static final int BUFFER_SIZE = 4;
    protected static int[] nums;
    protected static int counter;

    public ProducerConsumer()
    {
        nums = new int[BUFFER_SIZE];
        counter = 0;
    }

    protected synchronized void increment()
    {
        counter++;
    }

    protected synchronized void decrement()
    {
        counter--;
    }
}
```

```
public class Producer extends ProducerConsumer
{
  public void run()
  {
    int in = 0;
    for (int i = 1; i <= REPETITIONS; i++) {
      while (counter == nums.length) { }

      nums[in] = i;
      in = (in+1) % nums.length;

      System.out.println("Producer: " + i);
      try { … } catch ( … ) { }

      increment();
    }
  }
}
```

```
public class Consumer extends ProducerConsumer
{
  public void run()
  {
    int out = 0, sum = 0;
    for (int i = 1; i <= REPETITIONS; i++) {
      while (counter == 0) { }

      sum += nums[out];
      out = (out+1)%nums.length;

      System.out.println("Consumer: " + sum);
      try { … } catch ( … ) { }

      decrement();
    }
  }
}
```

16

Semaphores

a semaphore is a general purpose synchronization tool

- can think of semaphore as an integer variable, with two atomic operations
- in pseudocode (with each test/incr/decr being atomic):

```
void wait(Sempahore & S) {
    while (S <= 0) {}
    S--;
}
```

a.k.a *spinlock*

```
void signal(Semaphore & S) {
    S++;
}
```

N-process critical section problem

```
Pi:
 while (true) {
   wait(flagSemaphore);

   critical section

   signal(flagSemaphore);

   remainder section
 }
```

force Part1 to execute before Part2

```
P0:  ...
     Part1;
     signal(synchSemaphore);
     ...

P1:  ...
     wait(synchSemaphore);
     Part2;
     ...
```

17

Waitless semaphores

semaphores can be implemented so that busy waiting is avoided

- treat a semaphore as a resource, like an I/O device
- if the semaphore is not positive (not available), then rather than actively waiting for the resource, *block* the process so that it gives up the CPU
- a waiting process can be unblocked when the semaphore is available

semaphore is a structure that contains the int value & a queue of waiting processes

```
struct Semaphore {
    int value;
    Queue<PCB> queue;
};
```

remember: wait & signal must be atomic operations

```
void wait(Semaphore S) {
  S.value--;
  if (S.value < 0) {
      S.queue.Enter(currentProcess);
      block(currentProcess);
  }
}

void signal(Semaphore S) {
  S.value++;
  if (S.value <= 0) {
      S.queue.Remove(waitingProcess);
      wakeup(waitingProcess);
  }
}
```

18

Classic problem: bounded buffer

recall: producer writes to shared buffer (size N), consumer accesses data

- can be solved using semaphores (`full` is initially 0, `empty` is initially N)

Producer

```
in = 0;
while (true) {
    PRODUCE nextP

    wait(empty);
    wait(mutex);

    buffer[in] = nextP;
    in = (in + 1) % BUFFER_SIZE;

    signal(mutex);
    signal(full);
}
```

Consumer

```
out = 0;
while (true) {
    wait(full);
    wait(mutex);

    nextC = buffer[out]
    out = (out + 1) % BUFFER_SIZE;

    signal(mutex);
    signal(empty);

    CONSUME nextC
}
```

19

Concrete Java example

Java provides a `Semaphore` class – can utilize and avoid the counter and flags

```
public class ProducerConsumer extends Thread
{
    protected static final int REPETITIONS = 10;
    protected static final int BUFFER_SIZE = 4;
    protected static int[] nums;
    protected static Semaphore empty, full, mutex;

    public ProducerConsumer()
    {
        nums = new int[BUFFER_SIZE];

        empty = new Semaphore(BUFFER_SIZE);
        full = new Semaphore(BUFFER_SIZE);
        full.drainPermits();
        mutex = new Semaphore(1);
    }
}
```

```
public class Producer extends ProducerConsumer
{
  public void run()
  {
    int in = 0;
    for (int i = 1; i <= REPETITIONS; i++) {
      try {
        empty.acquire();
        mutex.acquire();

        nums[in] = i;
        in = (in+1) % nums.length;
        System.out.println("Producer: " + i);

        mutex.release();
        full.release();
      }
      catch (InterruptedException e) { }
    }
  }
}
```

```
public class Consumer extends ProducerConsumer
{
  public void run()
  {
    int out = 0, sum = 0;
    for (int i = 1; i <= REPETITIONS; i++) {
      try {
        full.acquire();
        mutex.acquire();

        sum += nums[out];
        out = (out+1) % nums.length;
        System.out.println("Consumer: " + sum);

        mutex.release();
        empty.release();
      }
      catch (InterruptedException e) { }
    }
  }
}
```

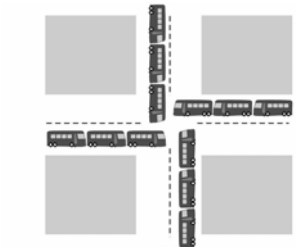
20

Deadlock

when multiple semaphores are used, *deadlock* is a potential problem

- two or more processes waiting indefinitely for an event that can be caused by only one of the waiting processes

```
P0:  ...                    P1:  ...
     wait(S);                    wait(Q);
     wait(Q);                    wait(S);
     ...                         ...
     signal(S);                  signal(Q);
     signal(Q);                  signal(S);
     ...                         ...
```



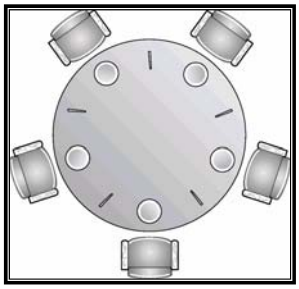
deadlock is a danger whenever processes require multiple shared resources

techniques for avoiding/handling deadlock are described in Ch. 7

21

Classic problem: dining philosophers

N philosophers are sitting at a round table in a Chinese restaurant with a single chopstick in between each of them. How do they eat?



```
Philospher_i:

  while (true) {
      wait(chopstick[i]);
      wait(chopstick[(i+1)%N]);

      EAT

      signal(chopstick[(i+1)%N]);
      signal(chopstick[i]);
}
```

to handle deadlock, could

- allow at most N-1 philosophers at the table
- require grabbing both chopsticks simultaneously (in critical section)
- add asymmetry (odd numbered philosopher grabs left first, even grabs right)

22

Critical regions & monitors

while semaphores work, it is possible to have problems if used incorrectly

e.g., wrong order (signal followed by wait)
wrong function call (wait followed by wait)
missing function (wait but no signal)

other high-level synchronization constructs (built on top of semaphores) are provided by various languages/applications

- a *critical region* is a synchronization construct that controls access to code blocks

```
in = 0;
while (true) {
  PRODUCE nextP

  region buffer when count < N do
  {
    buffer[in] = nextP;
    in = (in + 1) % BUFFER_SIZE;

    count++;
  }
}
```

```
out = 0;
while (true) {
  region buffer when count > 0 do
  {
    nextC = buffer[out]
    out = (out + 1) % BUFFER_SIZE;

    count--;
  }

  CONSUME nextC
}
```

- a *monitor* is an object-oriented synchronization construct
  data fields define the state of the monitor
  methods implement operations (constrained so that only 1 can be active)

23

OS synchronization schemes

Solaris: provides real-time computing, multithreading, multiprocessing

- uses *adaptive mutexes* to protect access to short critical sections
  - adaptive mutex begins as a semaphore implemented as a *spinlock* (busy waiting)
  - if the resource is held by an executing thread, then the adaptive mutex continues as a spinlock (waits for the resource to become available)
  - if the resource is held by a non-executing thread (e.g., if a uniprocessor), adaptive mutex suspends itself
- for longer critical sections, utilizes semaphores & *condition variables* to suspend
- suspended threads are put in waiting queues, or *turnstiles*

Windows XP: similarly real-time, multithreading, multiprocessing

- on a uniprocessor system, uses *interrupt mask* to protect access
  when kernel accesses a shared resource, it masks (disables) interrupts for all event-handlers that may also access the resource
- on a multiprocessor system, uses *adaptive mutexes* and other constructs
  for efficiency, kernel threads can never be preempted while holding a spinlock

24

Tuesday: TEST 1

types of questions:

- factual knowledge: TRUE/FALSE
- conceptual understanding: short answer, discussion
- synthesis and application: process scheduling, C++ simulation, …

the test will include extra points (Mistakes Happen!)

- e.g., 52 or 53 points, but graded on a scale of 50

study advice:

- review online lecture notes (if not mentioned in class, won't be on test)
- review text
- reference other sources for examples, different perspectives
- look over quizzes

25