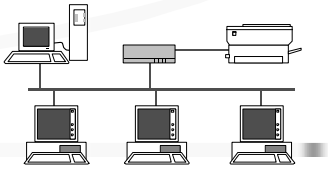


Operating Systems
600.418
Process Synchronization

Department of Computer Science
The Johns Hopkins University

Process Synchronization

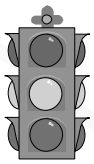
Lecture 3

Reading:
- Silberschatz & Galvin
- chapter 6

Additional Reading:
- Stallings
- chapter 5

The Problem with Concurrent Execution

- Concurrent processes (or threads) often need access to shared data and shared resources.
- If there is no controlled access to shared data, it is possible to obtain an inconsistent view of this data.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.



Simple Example

- Threads T1 and T2 are executing the same procedure using the same variables *in* and *out*.
- Threads can be interrupted anywhere.
- What can happen ?

```
Shared variables in, out;

Procedure Echo();
    read( in, keyboard );
    out := in;
    write( out, screen );
end Echo;
```

Simple Example (cont.)

- Threads T1 and T2 are executing the same procedure using the same variables *in* and *out*.
- Threads can be interrupted anywhere.
- Race condition - the output depends on the way the threads are interleaved.

```
Shared variables in, out;

Procedure Echo();
      read( in, keyboard );
      out := in;
      write( out, screen );
end Echo;
```

The Critical Section Problem

- *n* processes are competing to use some shared data.
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Goal: when one process is executing in its critical section, no other process can execute in its critical section.
- Each process looks like this:

```
Repeat
      entry section
            critical section - CS
      exit section
            remainder section - RS
Forever;
```

Requirements for Critical-Section Solutions

1. **Mutual Exclusion.** If process *Pi* is executing in its critical section (CS), then no other process can execute in its CS.
2. **Progress.** If no process is executing in its CS and there exist some processes that wish to enter their CS, then the selection of the process that will enter the CS next cannot be postponed indefinitely.
3. **Bounded Waiting.** There exist a bound on the number of times that other processes are allowed to enter their CS after a process has made a request to enter its CS and before that request is granted.

Types of Solutions

- **Software solutions**
  - **Algorithms whose correctness does not rely on any assumptions other than positive processing speed (that may mean no failure).**
  - **Busy waiting.**
- **Hardware solutions**
  - **Rely on some special machine instructions.**
- **Operating system solutions**
  - **Extending hardware solutions to provide some functions and data structure support to the programmer.**

Software Solution 1 (incomplete)

- Only two processes.
- Local process is *i*, other is *j*.
- What is good ?
- What is bad ?

```
Shared turn: (0..1)=0;

Repeat
    while turn ≠ i do no-op;
        critical section
    turn := j;
        remainder section
Forever
```


Software Solution 1 (incomplete)

- Only two processes.
- Local process is *i*, other is *j*.
- Mutual exclusion - **good**.
- Progress - **bad**.
- Bounded time - well, if progress was guaranteed, bounded time was ok too.

```
Shared turn: (0..1)=0;

Repeat
    while turn ≠ i do no-op;
        critical section
    turn := j;
        remainder section
Forever
```

Software Solution 2 (incomplete)

- Only two processes.
- Local process is *i*, other is *j*.
- What is good ?
- What is bad ?

```
Shared flag: array [0..1] of
    boolean=[false, false];

Repeat
    flag[ i ] := true;
    while flag[ j ] do no-op;
        critical section
    flag[ i ] := false;
        remainder section
Forever
```


Software Solution 2 (incomplete)

- Only two processes.
- Local process is *i*, other is *j*.
- Mutual exclusion - **good**.
- Progress - **bad**. If both executed the second line before any of them entering the while statement, we have a problem.
- Bounded time - irrelevant.

```
Shared flag: array [0..1] of
    boolean=[false, false];

Repeat
    flag[ i ] := true;
→   while flag[ j ] do no-op;
        critical section
    flag[ i ] := false;
        remainder section
Forever
```

Software Solution 3 (Peterson's)

- Only two processes.
- Local process is *i*, other is *j*.
- Mutual exclusion - **good**.
- Progress - **good**.
- Bounded time - **good**.

BUT: It only works for two processes!

```
Shared turn: (0..1)=0;
Shared flag: array [0..1] of
   boolean=[false, false];

Repeat
   flag[ i ] := true; turn := j;
   while (flag[ j ] and turn=j)
                  do no-op;
         critical section
   flag[ i ] := false;
         remainder section
Forever
```


The Bakery Algorithm

- Generalization for *n* processes.
- Each process has an id. Ids are ordered.
- Before entering its critical section, a process receives a number. The holder of the smallest number enters its critical section.
- Tie breaker is done using the process id: if processes *i* and *j* hold the same number and $i < j$ then *i* enters first.

The Bakery Algorithm (cont).

Shared *number*: array [1..n] of integer = [0,…, 0];
Shared *choosing*: array [1..n] of boolean=[*false*,..., *false*];

Repeat
  choosing[i] := true; *number[i] :=max(number*[1..n]) + 1;
  choosing[i] := false;
  for *j* := 1 to *n*
    while choosing[*j*] do no-op;
    while (*number[j] ≠ 0* and
      (*number[j], j*) ≺ (*number[i], i*))do no-op;
  end for;
    critical section
  number[i] := 0;
    remainder section
Forever

Hardware Solution - Disable Interrupts

- Correct solution for a uni-processor machine.
- During critical section multiprogramming is not utilized - performance penalty.
- On a multiprocessor machine the solution is not correct.

Repeat
  disable interrupts
    critical section
  enable interrupts
    remainder section
Forever

Hardware Solution - Test and Set

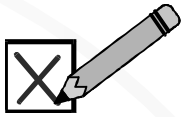
- Test and modify the content of a word **atomically**.

Function Test-and-Set(var target: boolean) : boolean
begin
    Test-and-Set := target;
    target := true;
end

- Other variants exist.

Shared *lock*: boolean =*false*;

Repeat
    while *Test-and-Set*(*lock)*
        do no-op*;*
    critical section
    lock := *false*;
    remainder section
Forever



Operating System Solution - Semaphore

- Synchronization tool that does not require busy waiting.
- Can only be accessed via two **atomic** operations, which **logically** look like this:

***s* : integer**

***wait*(*s*): while *s* ≤ 0 do no-op**
        ***s* := *s* - 1;**

***signal*(*s*): *s*:= *s* + 1;**

mutex : semaphore :=1;

Repeat
    wait(*mutex*);
        critical section
    signal(*mutex*);
        remainder section
Forever

Semaphore Implementation

- Semaphore is a record:

  type *semaphore* = record
  value: integer;
  L : list of *process*;
  end;

- Assumes two operations:
  - ***block*** suspends the process that invokes it.
  - ***Wakeup*(*p*)** resumes the execution of a blocked process *p*.

Atomic operations:

- *Wait(s):*

  S.value := S.value -1;
  if S.value < 0
  add this process to S.L;
  block;
  end;
- *Signal(s):*

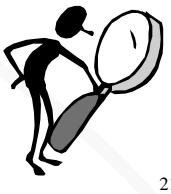
  S.value := S.value + 1;
  if S.value ≤ 0
  remove a process *P* from S.L;
  wakeup(P);
  end;

Semaphore Properties

- **S.value >= 0 :** value is the number of additional processes that can execute wait(s) without blocking.
- **S.value < 0 :** | value | is the number of processes already blocked.
- **Atomicity**: wait(s) and signal(s) are atomic also in the sense that no two processes can be inside wait(s) and signal(s) at the same time (not even one in wait and one in signal, even with multiple CPUs).
- Practical implementation of the wait/signal critical section:
  - Uniprocessor: disable interrupts.
  - Multiprocessor: software or hardware schemes.

Semaphore Types

- Counting Semaphore - the integer value can be any integer.
- Binary Semaphore - the integer value can be either 0 or 1.
- Binary semaphore might be easier to implement.
- Counting semaphore can be implemented using binary semaphore. HOW ?



From Binary Semaphores to Counting Semaphores

BS1 : binary semaphore = 1;
BS2 : binary semaphore = 0;
Count: integer = initial value of counting semaphore.

- Wait(s)

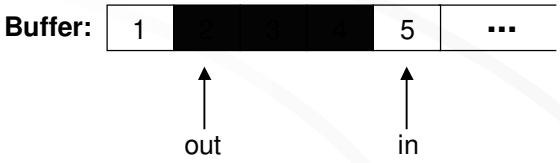
```
wait( BS1 ); Count := Count - 1;
if Count < 0
        signal (BS1 ); wait( BS2 );
signal ( BS1 );
```

- Signal (s)

```
wait( BS1 ); Count := Count + 1;
if Count ≤ 0 then signal( BS2 ) else signal( BS1 );
```

The Producer / Consumer Problem

- Also called the bounded-buffer problem.
- A producer produces data that is consumed by a consumer. (e.g. spooler and printer).
- A buffer holds the produced data which is not yet consumed.
- There exist several producers and consumers
- Initially, we assume the buffer is unbounded.



Producer / Consumer : unbounded buffer

Initialization: semaphores: mutex = 1; Full = 0;
integers : in = 0; out = 0;

```
append
    buffer[in]:= indata;
    in := in + 1;
end append;

take
    outdata:=
    buffer[out];
    out := out + 1;
end take;
```

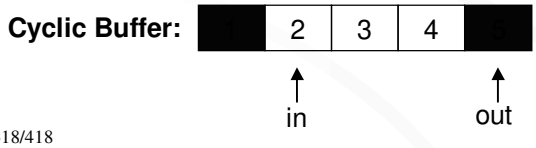
```
Producer                    Consumer

Repeat                      Repeat
        produce                     wait( Full );
indata;                             wait( mutex );
        wait( mutex );              take;
        append;                     signal( mutex
        signal( mutex       );
);                                  consume
        signal( Full );     outdata;
Forever                     Forever
```

Does it matter to switch the two signals in the producer ?

Does it matter to switch the two waits in the consumer ?

Producer / Consumer with Bounded Buffer

- **As before**, the consumer can take when the number of data items in the buffer is at least one.
- **This time**, the producer can append when the number of empty cells in the buffer is at least one.
- Can you modify the previous code to accommodate a bounded buffer ?
- **The trick is to realize that the consumer actually produces empty cells!**



Producer / Consumer : Bounded Buffer

Initialization: semaphores: mutex = 1; Full = 0; Empty = N;
integers : in = 0; out = 0;

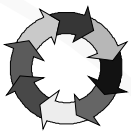
```
append
    buffer[in]:= indata;
    in := ( in + 1 ) mod N;
end append;

take
    outdata:= buffer[out];
    out := (out + 1) mod N;
end take;
```

Producer

```
Repeat
        produce indata;
        wait( Empty );
        wait( mutex );
        append;
        signal( mutex );
        signal( Full );
Forever
```

Consumer

```
Repeat
        wait( Full );
        wait( mutex );
        take;
        signal( mutex );
        signal( Empty );
        consume
outdata;
Forever
```



The Readers / Writers Problem

- **Writing is exclusive.**
- **Several readers can read together.**
- Readers have priority over writers.

Initialization

```
semaphores: mutex = 1;
            wrt = 1;
integer : rcount = 0;
```

Writer

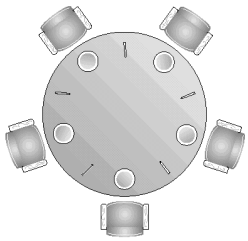
```
wait( wrt );
    …
    writing is performed
    …
signal( wrt );
```

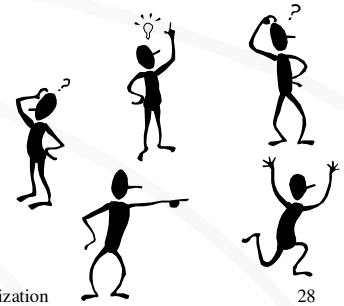
Reader

```
wait( mutex );
    rcount := rcount + 1;
    if (rcount = 1 ) then
    wait( wrt );
signal( mutex );
    …
reading is performed
    …
wait( mutex );
    rcount := rcount - 1;
    if (rcount = 0 ) then
        signal( wrt );
signal( mutex );
```


The Dining Philosophers

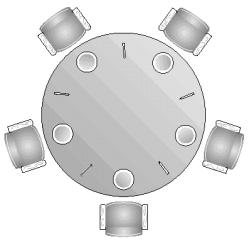
- Five philosophers who spend their lives thinking and eating.
- To eat, a philosopher needs two forks.
- A philosopher may only pick one fork at a time.
- **Think of each philosopher as a process and the forks are resources.**





The Dining Philosophers (cont.)

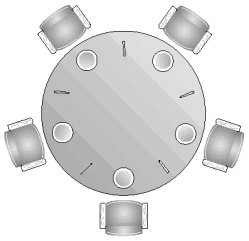
```
Array [1..5] of semaphores : fork = [1, 1, 1, 1, 1 ];

Philosopher Pi:
Repeat
    think;
    wait( fork[i] );
    wait( fork[ (i+1) mod 5 );
    eat;
    signal( fork[ (i+1) mod 5 );
    signal( fork[i] );
Forever
```



Well, what do you eat?
… sorry, …, think?

The Dining Philosophers (cont.)

```
Array [1..5] of semaphores : fork = [1, 1, 1, 1, 1 ];
semaphore : table = 4;
Philosopher Pi:
Repeat
    think;
    wait( table ); wait( fork[i] );
    wait( fork[ (i+1) mod 5 );
    eat;
    signal( fork[ (i+1) mod 5 );
    signal( fork[i] ); signal( table );
Forever
```



Allow only 4 of them to come to the table at the same time

Difficulties with Semaphores

Semaphores provide a powerful synchronization tool.

BUT:

- **wait(s) and signal(s) are scattered among several processes. Therefore, it is difficult to understand their effects.**
- **Usage must be correct in all the processes.**
- **One bad process (or one programming error) can kill the whole system.**

Monitors

- A monitor is a software module that contains:
  - **Procedures**
  - **Initialization code**
  - **Shared data**
- Monitor properties:
  - **Shared data can be accessed only by monitor's procedures.**
  - **Only one process at a time can execute in the monitor (executing a monitor procedure).**
- Shared data may contain condition variables.

Condition Variables in Monitors

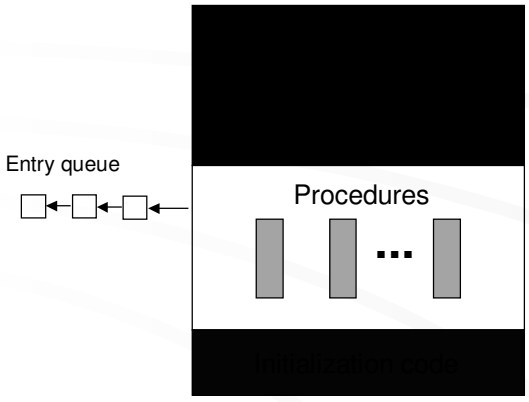
- To allow a process to wait within the monitor, a condition variable must be used.
- Condition variables can only be used with the following operations:
  - condition : x

    declaring a condition variable x
  - x.wait

    the process invoking **x.wait** is suspended until another process invokes **x.signal**
  - x.signal

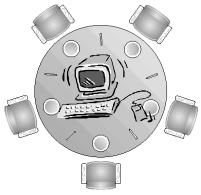
    resumes exactly one suspended process. If no process is suspended, this operation has no effect.

Monitor Architecture



Monitor for the Dining Philosophers

Monitor dining-philosophers

```
Array[1..5] of (think, hungry, eat) : State;
Array[1..5] of condition : Self;

procedure entry pickup( i )
    State[i] = hungry;
    test( i );
    if State[i]≠eat then Self[i].wait;

procedure entry putdown( i )
    State[i] = think;
    test( i-1 mod 5 );
    test( I+1mod 5 );

Initialization
    for i:= 1 to 5  state[i] = think;
```

```
procedure test( k )
    if State[k-1 mod 5] ≠ eat and
      State[k] = hungry and
      State[k+1 mod 5] ≠ eat then
          State[k] := eat;
          Self[k].signal;
    end if
```