Synchronization

(Chapter 2.3, Tanenbaum)

Concurrency

Motivation: overlap computation with I/O; simplify programming.

- **hardware parallelism**: CPU computing, one or more I/O devices are running at the same time.
- **pseudo parallelism**: rapid switching back and forth of the CPU among processes, pretending that those processes run concurrently.
- **real parallelism**: can only be achieved by multiple CPUs.

Single CPU systems cannot achieve real parallelism, but...

Keeping track of multiple activities is **difficult**.

Concurrent processes

In a multiprogramming environment, processes executing concurrently are either *competing* for the CPU and other global resources, or *cooperating* with each other for sharing some resources.

An OS deals with competing processes by carefully allocating resources and properly isolating processes from each other. For cooperating processes, on the other hand, the OS provides mechanisms to share some resources in certain ways as well as allowing processes to properly interact with each other.

Cooperation is either by implicit sharing or by explicit communication.

Processes: *competing*

Processes that do not exchange information cannot affect the execution of each other, but they can compete for devices and other resources. Such processes do not intend to work together, and so are unaware of one another.

Example: Independent processes running on a computer.

Properties:

- Deterministic.
- Reproducible.
- Can stop and restart without "side" effects.
- Can proceed at arbitrary rate.

Processes: *cooperating*

Processes that are aware of each other, and directly (by exchanging messages) or indirectly (by sharing a common object) work together, may affect the execution of each other.

Example: Transaction processes in airline reservations.

Properties:

- Share (or exchange) something: a common object (or a message).
- Non-deterministic (a problem!).
- May be irreproducible (a problem!).
- Subject to race conditions (a problem!).

Why cooperation?

We allow processes to cooperate with each other, because we want to:

- share some resources.
  - One checking account file, many tellers.
- do things faster.
  - Read next block while processing current one.
  - Divide jobs into smaller pieces and execute them concurrently.
- construct systems in modular fashion.

  UNIX example:

```
cat infile | tr ' ' '\012' |
  tr '[A-Z]' '[a-z]' | sort | uniq -c
```

A potential problem

Instructions of cooperating processes can be interleaved arbitrarily. Hence, the order of (some) instructions are irrelevant. However, certain instruction combinations must be eliminated. For example:

Process A	Process B	concurrent access
`A = 1;`	`B = 2;`	*does not matter*
`A = B + 1;`	`B = B * 2;`	*important!*

A *race condition* is a situation where two or more processes access shared data concurrently **and** correctness depends on specific interleavings of operations (i.e. good luck).

An example

time	**Person A**	**Person B**
3:00	Look in fridge. *Out of milk*.	
3:05	Leave for store.	
3:10	Arrive at store.	Look in fridge. *Out of milk*.
3:15	Buy milk.	Leave for store.
3:20	Leave the store.	Arrive at store.
3:25	Arrive home, put milk away.	Buy milk.
3:30		Leave the store.
3:35		Arrive home. ***OH! OH!***

What does correct mean? *Someone gets milk, but NOT everyone (too much milk!)*

Mutual exclusion

The "too-much-milk" example shows that when cooperating processes are not synchronized, they may face unexpected "timing" errors.

Mutual exclusion is a mechanism to ensure that only one process (or person) is doing certain things at one time, thus avoid data inconsistency. All others should be prevented from modifying shared data (i.e., the fridge or milk) until the current process finishes.

E.g., only one person *buys milk* at a time.

Critical section

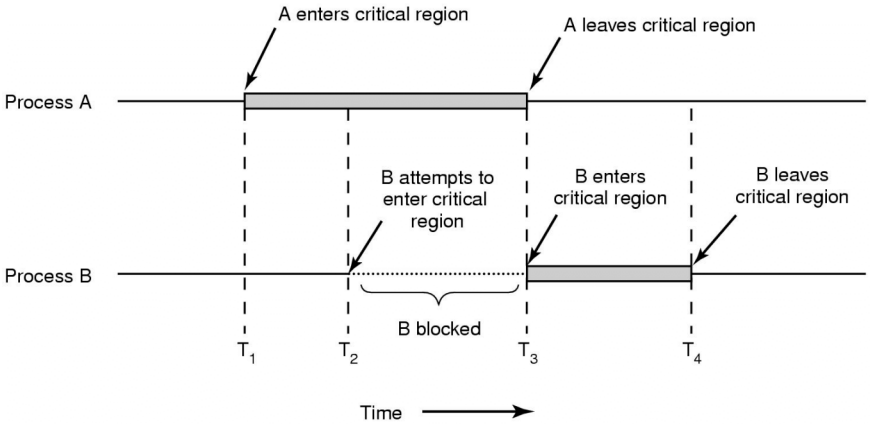
A *section of code*, or a *collection of operations*, in which only one process may be executing at a given time and which we want to make "sort of" atomic. *Atomic* means either an operation happens in its entirety or NOT at all; i.e., it cannot be interrupted in the middle.

E.g., *buying milk* or *shopping*.

Atomic operations are used to ensure that cooperating processes execute correctly.

Mutual exclusion mechanisms are used to solve the *critical section* problem.

*Critical Regions (2)



Mutual exclusion using critical regions

Solution 1

First attempt at computerized milk buying:

Processes A & B

```
if ( NoMilk ) {
   if ( NoNote ) {
      Leave Note;
      Buy Milk;
      Remove Note;
   }
}
```

This solution works for people because the first three lines are performed atomically*; but does not work otherwise.*

Solution 2

Second attempt: use 2 notes.

Process A

```
Leave NoteA;
if ( NoNoteB ) {
   if ( NoMilk ) {
      Buy Milk;
   }
}
Remove NoteA;
```

Process B

```
Leave NoteB;
if ( NoNoteA ) {
   if ( NoMilk ) {
      Buy Milk;
   }
}
Remove NoteB;
```

What can you say about this solution?

Solution 3

Third attempt: in case of tie, B will buy milk.

Process A

```
Leave NoteA
if ( NoNoteB ) {
   if ( NoMilk ) {
      Buy Milk;
   }
}
Remove NoteA;
```

Process B

```
Leave NoteB;
while ( NoteA )
   ; // do nothing
if ( NoMilk ) {
   Buy Milk;
}
Remove NoteB;
```

This “asymmetric” solution works. But...

Critique for solution 3

The previous solution to the *too-much-milk* problem is much too complicated. The problem is that the mutual exclusion idea is simple-minded. Moreover, the code is asymmetric (and complex) and process B is consuming CPU cycles while waiting.

In any case, the solution would be even more complicated if extended to many processes (try to modify the code for 4 processes).

Fundamental requirements

Concurrent processes should meet the following requirements in order to cooperate correctly and efficiently using shared data:

1. *Mutual exclusion*—no two processes will simultaneously be inside the same critical section (CS).
2. *Progress*—a process wishing to enter its CS will eventually do so in finite time.
3. *Fault tolerance*—processes failing outside their CS should not interfere with others accessing the CS.
4. *No assumptions*—should be made about relative speeds or the number of processors. Must handle all possible interleavings.
5. *Efficiency*—a process will remain inside its CS for a short time only, without blocking.

Also, a process in one CS should not block others entering a different CS.

Mutual exclusion—attempt 1

Process A

```
...
while( TRUE ) {
  ...
  while( proc == B ) ;
   < critical_A >
  proc = B;
  ...
}
...
```

Process B

```
...
while( TRUE ) {
  ...
  while( proc == A ) ;
   < critical_B >
  proc = A;
  ...
}
...
```

proc is a global variable and initialized to **A** (or **B**). Both processes start execution concurrently.

Problem: *violates rule 2 (strict alteration).*

Mutual exclusion—attempt 2

Process A

```
...
while( TRUE ) {
  ...
  while( pBinside ) ;
  pAinside = TRUE;
   < critical_A >
  pAinside = FALSE;
  ...
}
...
```

Process B

```
...
while( TRUE ) {
  ...
  while( pAinside ) ;
  pBinside = TRUE;
   < critical_B >
  pBinside = FALSE;
  ...
}
...
```

The global variables **pAinside** and **pBinside** are initialized to **FALSE**.

Problem: *violates rule 1 (interleaved instructions). Both A & B can be in the critical section.*

Mutual exclusion—attempt 3

Process A

```
...
while( TRUE ) {
  ...
  pAtrying = TRUE;
  while( pBtrying ) ;
   < critical_A >
  pAtrying = FALSE;
  ...
}
...
```

Process B

```
...
while( TRUE ) {
  ...
  pBtrying = TRUE;
  while( pAtrying ) ;
   < critical_B >
  pBtrying = FALSE;
  ...
}
...
```

The global variables **pAinside** and **pBinside** are renamed as **pAtrying** and **pBtrying**, respectively.

***Problem**: violates rule 2 (interleaved instructions).*

Dekker's algorithm

Process A

```
...
while( TRUE ) {
  ...
  pAtrying = TRUE;
  while( pBtrying )
    if( turn == B ) {
      pAtrying = FALSE;
      while( turn == B ) ;
      pAtrying = TRUE;
    }
  < critical_A >
  turn = B;
  pAtrying = FALSE;
  ...
}
...
```

Process B

```
...
while ( TRUE ) {
  ...
  pBtrying = TRUE;
  while( pAtrying )
    if( turn == A ) {
      pBtrying = FALSE;
      while( turn == A ) ;
      pBtrying = TRUE;
    }
  < critical_B >
  turn = A;
  pBtrying = FALSE;
  ...
}
...
```

One more global variable, **turn**, initialized to **A** or **B**.

Peterson's algorithm

Process A

```
...
while( TRUE ) {
   ...
   pAtrying = TRUE;
   turn = B;
   while( pBtrying &&
          turn == B ) ;
    < critical_A >
   pAtrying = FALSE;
   ...
}
...
```

Process B

```
...
while( TRUE ) {
   ...
   pBtrying = TRUE;
   turn = A;
   while( pAtrying &&
          turn == A ) ;
    < critical_B >
   pBtrying = FALSE;
   ...
}
...
```

Same global variables, but **turn** need not be initialized *(Note: the above relies on a race condition to resolve access rights, but it is not harmful).*

Yes, correct; but...

Both Dekker's and Peterson's algorithms are correct. However, they only work for 2 processes. Similar to the last solution of the "*too-much-milk*" problem, these algorithms can be generalized for N processes, but:

- N must be fixed (known *a priori*), which is too much to expect.
- Again, the algorithms become much too complicated and expensive.

Implementing a mutual exclusion mechanism is **difficult!**

Bakery algorithm

Process$_i$

```
boolean choosing[n];
int number[n];
...
while( TRUE ) {
  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],j) < (number[i],i) ) ;
  }
   < critical_i >
  number[i] = 0;
  ...
}
```

Where:

- *max(a_0, ..., a_{n-1}) is a number, k, such that k>=a_i for i=0, ..., n-1*, and
- ***(a,b) < (c,d) if a<c or if a=c and b<d***.

What is missing?

"*Real*" solutions are based on stronger prerequisites (e.g., fairness) than just variable sharing. Besides, we don't want to guess what is "atomic" when programming. So, we want:

- Hardware support—special instructions.
- OS kernel provided synchronization primitives.

At the lowest level (hardware), there are two elementary mechanisms:

- interrupt masking
  - can only do inside OS; only on uniprocessor
- atomic read-modify-write
  - can do inside OS and at user-level; OK for multiprocessors

A simple minded alternative

Let's start by using hardware instructions to mask interrupts. If we don't let the CPU interrupt (i.e., take away the control from) the current process, the solution for N processes would be as simple as below:

Process$_i$

```
...
while( TRUE ) {
  disableInterrupts();
   < critical_i >
  enableInterrupts();
  ...
}
```

Unfortunately, there is only one system-wide critical section active at a time. Besides, no OS allows user access to privileged instructions!

Hardware support

Many CPUs today provide hardware instructions to read, modify, and store a word *atomically*. Most common instructions with this capability are:

- **TAS**—test-and-set (Motorola 68K)
- **CAS**—compare-and-swap (IBM 370 and Motorola 68K)
- **XCHG**—exchange (x86)
- **LL/SC** — load-linked/store-conditional (MIPS, PowerPC)

The basic idea is to be able to read out the contents of a variable (memory location), and set it to something else all in one execution cycle. Hence not interruptible. The use of these special instructions makes life easier!

Yet another alternative!

Process$_i$

```
...
while( TRUE ) {
  while( TAS(&guard) )
     ; // busy wait
   < critical_i >
  guard = FALSE;
  ...
}
```

TAS implementation

```
boolean TAS (int *flag)
{ boolean result;

  result = *flag;
  *flag = TRUE;
  return result;
}
```

+ Only one global guard variable is associated with each critical section (i.e., there can be many critical sections).
+ N processes; processes are unaware of N.
– Busy waiting!

Semaphores

A **semaphore** is a synchronization variable (guard) that takes on non-negative integer values with only two atomic operations:

```
P(semaphore): { while ( semaphore == 0 ) ;
Wait(semaphore)    semaphore-- }
```

PROBEREN
probe/test
"wait"

```
V(semaphore): { semaphore++ }
Free(semaphore)
```

VERHOGEN
release
"make higher"

Semaphores are simple, yet elegant, and allow the solution of many interesting problems. They are useful for more than just mutual exclusion.

Semaphore solution

Here is a solution of "too-much-milk" problem with semaphores:

Processes A & B

```
1    Wait(OKToBuyMilk);
2    if ( NoMilk ) {   //
3       Buy Milk;      // critical section
4    }                 //
5    Free(OKToBuyMilk);
```

Note: Semaphore `OKToBuyMilk` must initially be set to 1. Why?

Properties of semaphores

Semaphores are not provided by hardware, but they have several attractive properties:

- Simple.
- Work with many processes—*single resource serialization*.
- Can have many different critical sections with different semaphores.
- Each critical section has unique access semaphore.
- Can permit multiple processes into the critical section at once, if desirable—*multiple (identical) resources*.

However, they are unstructured and do not support data abstraction (*see* monitors). Unstructured => is a convention that relies on correct programming.

Possible uses of semaphores

- *Mutual exclusion.*
  - initialize the semaphore to one.
- *Synchronization of cooperating processes (signaling).*
  - initialize the semaphore to zero.
- Managing multiple instances of a resource.
  - initialize the semaphore to the number of instances.

Type of semaphores

Semaphores are usually available in two flavors:

- **binary**—is a semaphore with an integer value of 0 and 1.
- **counting**—is a semaphore with an integer value ranging between 0 and an arbitrarily large number. Its initial value might represent the number of units of the critical resources that are available. This form is also known as a **general** semaphore.

Implementation of semaphores

No existing hardware implements **Wait** and **Free** operations directly. So, semaphores must be built up in software using some other elementary synchronization primitive(s) provided by hardware.

Uniprocessor solution: can *disable interrupts* or hardware support

Multiprocessor solution: *harder!* Possibilities:

- Turn off *all* other processors (*not practical!*)
- Use hardware support for atomic operations.

Binary semaphores—*busy wait*

```
WaitB(int &s)
{
  while ( TAS(s) );
  return;
}
```

```
FreeB(int &s)
{
  s = FALSE;
  return;
}
```

Spinlock—the process "spins" while waiting for the "lock".

- potential indefinite postponement
- low efficiency (busy waiting)

Binary semaphores—*NON-busy wait*

```
typedef struct {
    boolean guard = FALSE;
    boolean flag = TRUE;
    Queue waitQ;
} SemaphoreB;
```

```
WaitB(SemaphoreB* s)
{
  while( TAS(s->guard) ) ;
  if( s->flag == TRUE ) {
     s->flag = FALSE;
     s->guard = FALSE;
  } else {
     waitOn(s->waitQ);
     reset guard; //tricky!
  }
  return;
}
```

```
FreeB(SemaphoreB* s)
{
  while( TAS(s->guard) ) ;
  if( waiting(s->waitQ) ){
     moveToReady(s->waitQ);
  } else {
     s->flag = TRUE;
  }
  s->guard = FALSE;
  return;
}
```


Counting semaphores

```
typedef struct {
   int count;
   SemaphoreB lock, delay;
} SEMAPHORE;
```

```
Wait(SEMAPHORE* s)
{
  WaitB(s->lock);
  s->count--;
  if ( s->count < 0 ) {
     FreeB(s->lock);
     WaitB(s->delay);
  } else
     FreeB(s->lock);
}
```

```
Free(SEMAPHORE* s)
{
  WaitB(s->lock);
  s->count++ ;
  if ( s->count <= 0 )
     FreeB(s->delay);
  FreeB(s->lock);
}
```

Limitations of (classic) semaphores

Typical problems with the use of semaphores:

- Their use is *NOT* enforced, but is by convention only.
- The operations do not allow a test for busy without a commitment to blocking.
- There are no additional arguments for Wait and Free operations.
- With improper use, a process may block indefinitely.
- There is no means by which one process can control another by using semaphores, without the cooperation of the controlled process.

So, people continued looking for alternatives...

Primitives revisited!

The synchronization primitives discussed so far allow us to access shared data as follows:

```
entry protocol
  < access shared data >
exit protocol
```

Semaphores give us some abstraction: the *protocol* access to shared data is transparent to the user (hidden in the implementation).

Now we are ready for even more abstraction...

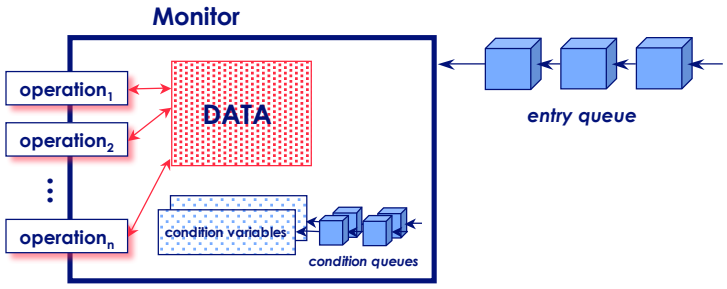
Monitors

A monitor is a high-level (programming language) abstraction that combines (and hides) the following:

- shared data
- operations on the data
- synchronization with condition variables

Mutual exclusion is **not** sufficient for concurrent programming. An additional way to block processes (e.g., when resource busy or buffer full) is also needed. Monitors use *condition variables* (cf. binary semaphores) to provide user-tailored synchronization and manage each with a separate condition queue. The only operations available on these variables are WAIT and SIGNAL.

Monitor abstraction



Compare monitors with objects (especially Java's synchronized methods).

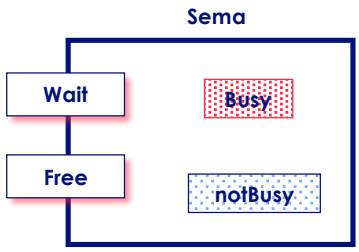
Equivalence of primitives

```
type Sema = monitor
  Busy: boolean;
  notBusy: condition;

procedure Wait()
{
  if ( Busy )
    WAIT.NotBusy;
  Busy = TRUE;
}

procedure Free()
{
  Busy = FALSE;
  SIGNAL.NotBusy;
}

begin
  Busy = FALSE;
end.
```



example:

```
Sema A;

  A.Wait();
  ...
  ...
  A.Free();
```

Semaphore implementation with monitors

Equivalence of primitives *cont.*

semaphore
- **lock** : serialize access to operations in monitor (= 1).
- **signal** : suspend processes executing **SIGNAL** (= 0).
- **condition**$_i$: suspend a process executing **WAIT** (= 0).

int
- **next** : number of processes waiting due to **SIGNAL**s (= 0).
- **cond-cntr**$_i$: number of processes waiting due to waiting on condition$_i$.

```
Operation_i()
{
  Wait(lock);

   < operation body >

  if ( next > 0 )
    Free(signal);
  else
    Free(lock);
}
```

```
condition_i.WAIT
{
  cond-cntr_i++;
  if ( next > 0 )
    Free(signal);
  else
    Free(lock);
  Wait(condition_i);
  cond-cntr_i--;
}
```

```
condition_i.SIGNAL
{
  next++;
  if ( cond-cntr_i > 0 )
    Free(condition_i);
  else
    Wait(signal);
  next--;
}
```

Process waits here.

Monitor implementation with semaphores

Other high-level primitives

There are several other proposed primitives for synchronization. The following are the most common mechanisms:

- Critical regions
- Conditional critical regions
- Eventcounts
- Sequencers
- Path expressions
- Serializers

These primitives are semantically equivalent. Moreover, any one can be built using the others. They are essentially provided by the systems software (OS kernel or language compilers) as programming tools.

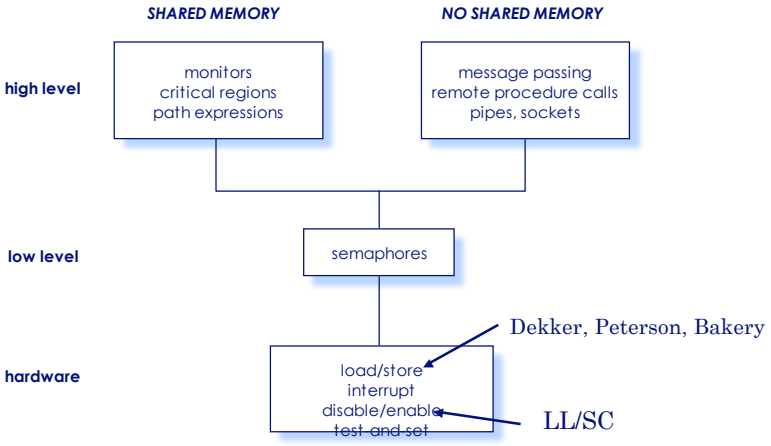
Problems with *synch* primitives

Livelock, or *starvation*: the situation in which some processes are making progress toward completion but some others are locked out of the resource(s). (cf. unfairness)

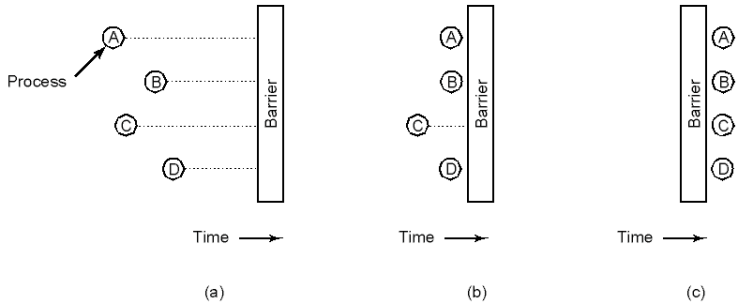
Deadlock: the situation in which two or more processes are locked out of the resource(s) that are held by each other.

The most important deficiency of the synchronization primitives discussed so far is that they were all designed on one or more CPUs accessing a "common" memory. Hence, these primitives are not applicable to distributed systems. **Solution**? Message passing...

Synch primitives—summary



*Barriers



Use of a barrier

- processes approaching a barrier
- all processes but one blocked at barrier
- last process arrives, all are let through

Transactions

A transaction is a "large" atomic operation, terminated by either a *commit* (successful termination) or an *abort* (unsuccessful termination) operation.

Since an aborted transaction may already have modified the various data it has accessed, the state of the data may not be the same as it would be after a successful (committed) transaction.

In this case, a transaction has to be "rolled back" (i.e., restored the modified data back to its state before the transaction started).

This topic is covered in database courses.