# Introduction to Processes and Threads

**Readings:** Tanenbaum, 2.1-2.2.4.

## *Sample questions that this topic addresses:*

What is a process and why is it a useful concept?

What is a thread and why is it useful?

What is the difference between a process and a thread? Why two concepts?

Uniprogramming and multiprogramming

Process state versus thread state

Examples from OSes

Keeping track of processes and threads

Running multiple threads on a single CPU

## *Processes*

We want to run multiple jobs at the same time on a system. Historically, it was to make better use of the CPU.

With many things happening at once in a system, need some way of separating them all out cleanly so that they are protected from each other and appear to be independent activities. That is a *process*.

Important concept: decomposition. Given hard problem, chop it up into several simpler problems that can be solved separately.

What is a process?

An intuitive definition is just a running piece of code along with all the things that the code can affect or be affected by.

Key aspect of a process: Processes protected from each other. Each process is given the illusion that it has protected state. What constitutes process state?

Is a process the same as a program?

## *Threads*

- What happens if a process needs to do multiple things in parallel?
  - Editor waiting for input, while doing some formatting or backup in background.
  - A web browser fetching a document from a server while allowing user to interact with the displayed windows.
  - A web server handling multiple user requests in parallel.

A thread can touch all the address space of its process  no memory protection between threads within a process. This is normally good: we want threads to cooperate with each other.

- Why separate Threads and Processes?

- Creating threads: normally a create_thread() call that takes a function pointer and a parameter as an argument.

- Thread status:
    - Show a state diagram showing the state transitions between different status of a thread.
        - Created, Runnable, Running, Blocked, Done (Zombie), Done

- Thread State kept in *thread control block*
    - Some state shared by all threads in a process: global variables, address space, open file handles.

- Some state "private" to each thread  each thread has its own copy: program counter, registers, execution stack (What is the execution stack?).

- Note that there is no enforced protection between threads in the same process. A poorly-coded thread can overwrite the stack of another thread, for example, trashing it.

Example OSes

- MS/DOS, MacOS: 1 address space, one thread per address space
- Traditional Unix: multiple address spaces, one thread/address space
- Mach, Solaris, VMS, HPUX, NT: multiple address spaces, multiple threads per address space

# Inter-thread and Inter-process communication

## *Readings for this topic:*

Tanenbaum: Section 2.3.

## *Questions:*

Why do processes/threads need to communicate?

What are race conditions?

Why is atomicity important?

## *Independent Threads*

Independent thread: one that can't affect or be affected by the rest of the universe.

Its state isn't shared in any way by any other thread.

Deterministic:  input state alone determines results.

Reproducible.

Can stop and restart with no bad effects (only time varies). Example:  a program that sums the integers from 1 to 100.

## *Cooperating threads/processes:*

Machine must model the social structures of the people that use it.  People cooperate, so machine must support that cooperation.  Cooperation means shared state, e.g. a single file system shared among multiple user processes.

Cooperating threads are those that communicate via share state.

## Problem with cooperating threads:

- Results may be non-deterministic
- They may not be easy to reproduce. Debugging can be difficult

Example:

- one process writes ``ABC'' to the terminal,
- another process writes ``CBA'' to the same terminal.

What are the potential outputs on the terminal? Are these cooperating processes? How?

When discussing concurrent threads, a single CPU vs. multiple CPU does not matter. Both lead to equally unpredictable results in the above example, assuming that, with a single CPU, the context-switch among threads can occur at any time.

- A given thread runs on only one processor at a time.

- A thread may run on different processors at different times (move state, assume processors are identical).

- Cannot distinguish multiprocessing from multiprogramming on a very fine grain.

Basic assumption for multi-threaded programs is that the order of some operations is irrelevant; certain operations are independent of certain other operations. Only a few things matter:

Example: A = 1; B = 2; has same result as B = 2; A = 1;

Another example: A = B+1; B = 2*B. Results depend on the order of operations.

Another example: suppose A = 1 and A = 2 are executed in parallel. What can be the final value of A?

If the results of execution depend on the order of execution of threads, we have a ***race condition.***

Don't know what will happen; depends on which one goes fastest. What if they happen at EXACTLY the same time, with multiple CPUs? Can't tell anything without more information. Could end up with A=3!

**Atomic operations:**

Before we can say ANYTHING about parallel threads, we must know that some operation is atomic, i.e. that it either happens in its entirety without interruption, or not at all. Cannot be interrupted in the middle.

***Key point****: If you don't have an atomic operation, you can't make one. Fortunately, the hardware guys give us atomic ops.*

References and assignments are atomic in almost all systems. A=B will always get a good value for B, will always set a good value for A (but not necessarily true if A and B are arrays or records).

**Example 1:**

*Suppose, initially A = 2, B = 3;*

*Thread 1:*

```
A = B;
if (A == B) cout << "they are equal";
else "they are not equal";
```

```
Thread 2;

B = A+1;
```

*Can you predict what will be printed by Thread 1?*

*Can you state possible final values of A and B?*

*Which of the following (A, B) values are possible?*

*(3, 4); (4, 4); (3, 3)?*

That can really make debugging multi-threaded programs hard because, by looking at just one thread, we cannot say anything about the values of variables at any statement.

Of course, the above is not a very interesting example because the code does not seem to be doing anything useful. Consider the following examples where threads do need to cooperate more closely.

**Example 2:** bank transfer example. Two people use an ATM card for the same account to initiate a withdrawal of $100, almost simultaneously. A thread is created to execute each transaction. The account has $150 in it initially. What can potentially happen if the two threads execute the following code?

*Thread 1:*

```
if (savings > 100)

      savings = savings - 100;
```

*Thread 2:*

```
if (savings > 100)

      savings = savings - 100;
```

Is it possible that both people end up withdrawing $100? Is bank guaranteed to detect the problem?

**Example 3:** Computerized milk buying: Too much milk problem

| Anne | Bob |
|---|---|
| // look in fridge<br>if (no milk)<br>    // go to Kroger<br>    Buy Milk<br>Put milk in fridge | // look in fridge<br>if (no milk)<br>    // go to Kroger<br>    Buy Milk<br>Put milk in fridge |

```
The following execution is possible:
```

## *What is needed to address the above problems?*

If you have any atomic operation, we need to be able generate *higher-level atomic operations* and make parallel programs work correctly. This is the approach we'll take in this class.

**Example 2 revisited:**

*Thread 1:*

```
--- begin atomic operation ---
if (savings > 100)
      savings = savings - 100;
-- end atomic operation ---
```

*Thread 2:*

```
-- begin atomic operation ---
if (savings > 100)
      savings = savings - 100;
-- end atomic operation ---
```

Now, no multiple withdrawals will be possible.

Some terminology

- ***critical section:*** A section of code that only 1 thread can execute at once (e.g. the code). The code appears to execute atomically even though other threads may execute in parallel.

Critical sections have the following required property:

- *mutual exclusion:* Only only 1 thread must be allowed to execute in a critical section at a time (others must be prevented

from entering the critical section). Thus, only 1 thread accesses and updates the savings variable at a time.

- *vacation property:* If a thread is outside the critical section (i.e., on vacation), other threads should not be prevented from going into the critical section.

- *no speed assumption:* No assumptions should be made about relative speeds of threads or the number of CPUs.

- *A thread* should not have to have to wait forever to enter the critical section.

In the banking example, we want the two lines of code in the threads to form a critical section.

## Solution to the critical section problem: Locks

A *lock*

- prevent someone from doing something,

e.g., before shopping, leave a note on the fridge

Three elements of locking

0. must aqcuire lock before doing something (e.g. entering critical section, accessing shared data
0. must release lock (unlock) when done
0. must wait to acquire if someone else has it locked

**`Example 3 revisited:`**

In general, wherever shared variables are manipulated, you need to worry about using locks to make a set of statements atomic (i.e., turn them into a critical section)

## How to implement locks?:

Suppose that, for now, we restrict ourselves to only using atomic load and store operations as the building block. Let's discuss the computerized milk-buying problem.

*Too Much Milk: Solution Attempt #1*

```
    basic idea:

      1. leave a note if you're going to buy milk (remove
note when you

          return)

      2. don't buy if the other person left a note
```

**Example 3 revisited:**

Use a note. Wait until there is no note. Leave a note before entering the critical section.

Solution:

Does this work in the computerized world?

At this point, before reading on, you should try to improve the solution and make it correct.

Some possible solutions that you may come up with:

**Too-much-milk: Solution Attempt #2:**

What if we use two notes and make Anne and Bob hang around to make sure they enter the critical section?

| Anne | Bob |
| --- | --- |
| **NoteAnne = true;** | **NoteBob = true;** |
| **while(NoteBob) loop;** | **while (NoteAnne)loop;** |
| if (no milk) | if (no milk) |
| Buy Milk; | Buy Milk; |
| **NoteAnne = false;** | **NoteBob = false;** |

Are Critical section requirements satisfied?

**Too-much-milk: Solution Attempt #3: Strict alternation**

What if Anne and Bob alternate? Then there will be no deadlock.

| Anne | Bob |
|---|---|
| <br><br>**while(!AnneTurn) loop;**<br><br>if (no milk)<br><br>        Buy Milk;<br><br>**AnneTurn = false;** | **while (AnneTurn)loop;**<br><br>if (no milk)<br><br>   Buy Milk;<br><br>**AnneTurn = true;** |

The above enforces the mutual exclusion condition. (Prove it!)

Any problem with the solution?

**Too-much-milk: Solution Attempt #4: (Peterson's Solution)**

If we combine solution #2 and solution #3, we can come up with a correct solution. The idea is to break ties using turns only if we run into a deadlock-like situation.

| Anne | Bob |
|---|---|
| **NoteAnne = true;**<br><br>**turn = Anne;**<br><br>**while(NoteBob && turn == Anne) loop;**<br><br>if (no milk)<br><br>       Buy Milk;<br><br>**NoteAnne = false;** | **NoteBob = true;**<br><br>**turn = Bob;**<br><br>**while(NoteAnne && turn== Bob)loop;**<br><br>if (no milk)<br><br>   Buy Milk;<br><br>**NoteBob = false;** |

**proof of correctness:**

## Correct but the solution is ugly:

- complicated, hard to convince yourself that it's right
- while Bob or Anne are waiting, they consume CPU time in the while loop.

This is called **busy-waiting. Busy waiting is bad.**

- Hard to see how to generalize the solution to more than 2 threads.

Fortunately, there is a better way:

- have hardware provide better (higher-level) primitives than atomic load and store (we'll cover this next)
- the OS provides higher-level abstractions for implementing critical sections based on this new hardware support

e.g. lock/unlock

lock: atomically wait 'til lock is free, then grab it

unlock: release lock, let anyone who's waiting for it try

Lock acquire/release must be atomic. If two threads are waiting for the lock and both see it's free, only one grabs the lock.

locks make "Too Much Milk" really easy to solve

```
Anne or Bob
 ---------
Lock mutex; // global, shared lock to both Anne and Bob

mutex.lock();
if (noMilk) {
    buy milk
}
```

```
mutex.unlock()
```

# Semaphores

## *Questions:*

What are the requirements for a locking mechanism?

Waiting for conditions: why we need another kind of synchronization besides locks?

Semaphores: a single mechanism to implement both locks and conditional waiting

How to solve the Producer & Consumer (Bounded Buffer) problem using semaphores?

Briefly look at another example: Readers and Writers Problem

## *Motivation*

The too-much-milk solution is much too complicated. The problem is that the mutual exclusion mechanism was too simple-minded: it used only atomic reads and writes. This is sufficient, but unpleasant. It would be unbearable to extend that mechanism to many processes.

Let's look at more powerful, higher-level mechanisms and what they need to do.

## Implementing Critical Sections:

We need to be able to implement critical sections.

***Some requirements*** **for critical section mechanism:**

- *Mutual Exclusion:* Only one process should be executing in a critical section at a time.
- *Progress:* If several requests at once, must allow one process to proceed.
- *Vacation Property:* Processes must be able to go on vacation outside critical section.

***Desirable properties*** **for a mutual exclusion mechanism**:

- *Bounded waiting (no starvation):* there must exist a bound on the number of times that other processes are allowed to enter their

critical sections after a process has made a request to enter its critical section and before that request is granted.

- *Efficient:* don't use up substantial amounts of resources when waiting. E.g. no busy waiting.
- *Simple:* should be easy to use (e.g. just bracket the critical sections).

Rules for processes using the mechanism:

- Always lock before manipulating shared data.
- Always unlock after manipulating shared data.
- Do not lock again if already locked.
- Do not unlock if not locked by you
- Do not spend large amounts of time in critical section.

Having a lock primitive provided by the OS would address the problem.

```
      Lock mutex;   // global, shared lock to both Anne and
Bob


      mutex.lock();

      // start of critical section

      if (noMilk) {

          buy milk

      }

      // end of critical section

      mutex.unlock()
```

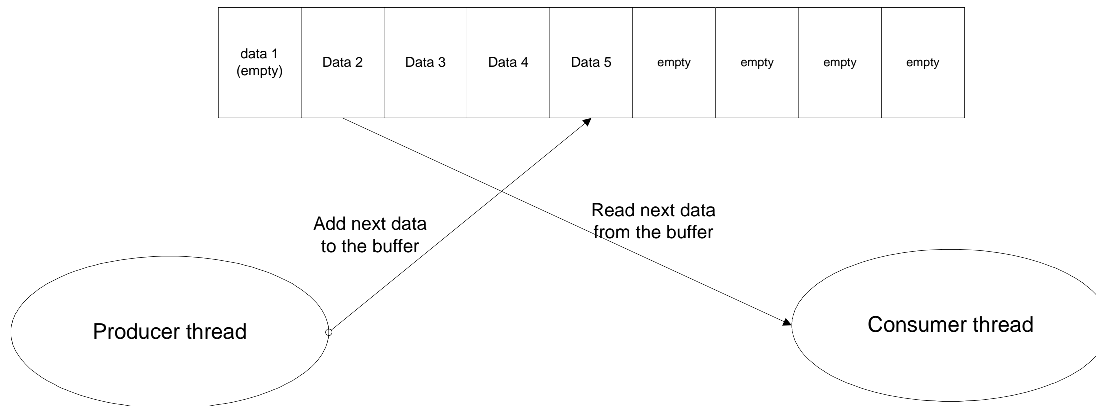
We will see later how to implement locks efficiently. (In fact, you will be doing that in Project 1!)

## *Another need for a mechanism: Scheduling*

Unfortunately, locks alone are not adequate for threads to work together. Consider the following problem. **This problem is important**:

## Producer & Consumer Problem (also called Bounded Buffer problem)

Often one thread creates information that needs to be used by another thread, as shown in the following picture:



Producer thread repeatedly adds items to the buffer. Consumer threads empties the buffer by reading items from it.

Examples of bounded buffer problem:

- Unix pipes:

```
tar cf - foo | gzip > foo.tar.gz
```

The tar process writes its output to a shared buffer. The gzip buffer reads from the buffer. There is one producer thread (tar) and one consumer thread (gzip)

- Servers:

One thread receives requests from clients and puts them in a buffer queue. One or more service threads grab requests from the queue and process them. There is one producer thread (one receiving the requests) and multiple consumer threads (the service threads that execute the requests).

- Printer spooler:

  Multiple processes send files to a printer. The files are queued in a spooler directory. A printer process grabs files from the queue and prints them. In this case, there are multiple producers (processes sending files to the printer queue) and one consumer (printer process).

In general, there could be multiple producer threads writing to the shared buffer and multiple consumer threads reading from the shared buffer, as illustrated in the above examples.

Also, in some cases, the shared buffer may be an unbounded queue, rather than a bounded queue.

Producer and consumer threads shouldn't have to operate in perfect lock-step: producer should be able to get ahead of consumer.

Correctness constraints:

- *Mutual exclusion:*

- *Scheduling:*

We can implement mutual exclusion using locks. How do we implement the scheduling constraints?

One possibility to address a scheduling constraint:

- If buffer is full, producers periodically check for a buffer to empty. This involves busy waiting – not very good.

A more efficient mechanism that has been proposed is **semaphores**. This was invented by Edsger Dijkstra, a Dutch computer scientist, in the mid 60's. Semaphores can be used to implement both locks and enforce scheduling constraints. A semaphore s is an OS-provided object that has a value >= 0 and provides two operations to threads:

- **s.down() or s.P():** The thread waits for semaphore to become positive, and then atomically decrements it by 1. (Proberen in dutch means down).
- **s.up() or s.V():** The thread atomically increments semaphore by 1 (verhogen in Dutch means up).

Note that if one or more threads is waiting for the semaphore value to become positive in s.down(), doing a s.up() operation will cause one of the threads executing the down operation to complete the down operation.

## Implementing mututal exclusion using semaphores:

Mutual exclusion can be easily implemented using semaphores. All we need is a semphore s that is initialized to the value 1.

- To acquire a lock: do a s.down()
- To release a lock: do a s.up()

### Too much milk problem with semaphores:

Show why there can never be more than one process buying milk at once.

## Bounded-buffer problem using semaphores:

You can find solution to the bounded-buffer problem in Tanenbaum. I will present a different solution here that may be conceptually simpler and easier to see how to apply it to other problems.

As mentioned earlier, there are three constraints to enforce: one mutual exclusion constraint and two scheduling constraints. Mutual exclusion constraint can be enforced using a binary semaphore *mutex* as in the too-much-milk problem.

To enforce the two scheduling constraints, we need to create one additional semaphore for each constraint:

- *OkToProduce:* This semaphore is initialized to number of empty locations in the buffer.
- *OkToConsume:* This semaphore is initialized to 0, i.e., number of full locations in the buffer initially.

The solution is as follows:

| Producer Threads | Consumer Threads |
| --- | --- |
| produceItem()<br><br>{<br><br>--- get an item to add ---<br><br>***OkToProduce.down();***<br><br>**mutex.down();**<br><br>buffer.add(item);<br><br>**mutex.up();**<br><br>***OkToConsume.up();***<br><br>} | consumeItem()<br><br>{<br><br>***OkToConsume.down();***<br><br>**mutex.down();**<br><br>item = buffer.deleteItem();<br><br>**mutex.up();**<br><br>***OkToProduce.up();***<br><br>--- consume the item ---<br><br>} |

Important questions (discussed in the lecture):

- Why does producer do OkToProduce.down() but OkToConsume.up()?

- Is the order of down's important?

- Is order of up's important?

- How would this be extended to have multiple consumers and multiple producers?

Semaphores aren't provided by hardware. (I'll describe implementation later) But they have several attractive properties:

- Machine independent.
- Simple.
- Work with many processes.
- Can have many different critical sections with different semaphores.
- Can acquire many resources simultaneously (multiple P's).
- Can permit multiple processes into the critical section at once, if that is desirable.

***Key Idea is layering:*** pick some powerful and flexible intermediate primitive to implement in the OS and apply to a large number of problems. The primitives in this case are the operations are:

- create a semaphore with an initial non-negative integer value.
- up operation on the semaphore
- down operation on the semaphore

Note that we do not need an operation to read or write the value of the semaphore. In fact, we do not want such an operation. Only up and down operations are sufficient for all thread synchronization needs.

## *Another Classical Problem: Readers and Writers Problem*

(this is optional for the lecture, but you should go through it and ask TAs to discuss it if needed)

Another example of semaphore usage: a shared database with readers and writers. It is safe for any number of readers to access the database simultaneously, but each writer must have exclusive access. Must use semaphores to enforce these policies. Example: checking account.

Note that writers are actually readers too.

*Scheduling constraints* that need to be enforced among threads:

- Writers can only proceed if there are no active readers or writers (use semaphore OKToWrite).

- Readers can only proceed if there are no active or waiting writers (use semaphore OKToRead).

Note that we don't want to make the access to the entire database as a critical section because then only one reader will be able to read at a time. We want multiple readers to be able read the database simultaneously – results in better response time for readers since a short read does not have to wait for a long read to complete.

To enforce scheduling constraints, we need to keep track of who's reading and writing. We thus introduce some shared variables to track that information:

- AR = number of active readers = 0 initially.

- WR = number of waiting readers = 0 initially.

- AW = number of active writers = 0 initially.

- WW = number of waiting writers = 0 initially.

Variables introduced to enforce scheduling constraints are sometimes called *state variables*. These variables will be read and written by multiple threads. We thus must introduce a **mutual exclusion** constraint to make sure that only one thread manipulates state variables at once (we will use a semaphore Mutex for this).

Initialization of semaphores:

- OKToRead = 0;
- OKToWrite = 0;
- Mutex = 1;

Our strategy in implementing readers and writers will be to always check the state variables before deciding to access the database. If it is not OK to access the database, then a reader will do a down operation on OkToRead and a write will do an up operation on OkToWrite.

When the read is completed by the last reader, the reader will be responsible for waking up one waiting writer, if there is one.

When the write is completed by a writer, it will wake up all waiting readers, allowing them to read the database.

The readers and writers execute the following code:

| Reader threads | Writer threads |
| --- | --- |
| get_read_access();<br><br>--- read data ---<br><br>release_read_access(); | get_write_access;<br><br>--- write data –<br><br>release_write_access(); |

No
te that access control on the database is enforced in the two procedures get_read_access() and get_write_access(). The other two methods are called by the threads to inform the enforcement mechanism that the operation is completed.

The code for the procedures is as follows:

```
get_read_access()
{
    Mutex.down();    // mutual exclusion constraint on state
variables
    if ((AW+WW) == 0) {
       AR = AR+1;        // a new reader being activated
       OkToRead.up(); // give advanced read permission (to self)
    else WR = WR+1;
    Mutex.up();          // end mutual exclusion constraint

    OkToRead.down();     // wait until permission granted
}
```

```
release_read_access()
{
    Mutex.down();          // enforce mutual exclusion constraint
    AR = AR1;             // one less active reader
    if (AR==0 && WW>0) {  // wake up a writer, if appropriate
         OKToWrite.up();
         AW = AW+1;       // important to remember that we woke
         WW = WW1;   // up a writer, before releasing mutex lock.
     }
     Mutex.up();           // finished updating state variables
 }

get_write_access()
{
      Mutex.down();   // mutual exclusion enforcement
      if ((AW+AR) == 0) {  // wait until no one else active
         AW = AW+1;
         OkToWrite.up();  // give advance permission to self
      }
      else WW = WW+1;    // tell others that we are waiting
      Mutex.down();   // done accessing state variables

      OkToWrite.down();  // wait until permission to write
}


release_write_acess()
{
     Mutex.down();
     AW = AW1;
     if (WW > 0) {
        OKToWrite.up();
        AW = AW+1;
        WW = WW1;
     }
     else {       // no writer activated. wake up waiting readers
        while (WR > 0) {
```

```
            OKToRead.up();  // wake up one waiting reader
            WR = WR1;
        }
      }
      Mutex.up();
}
```

Try going through several examples and figure out what happens:

- Reader enters and leaves system.
- Writer enters and leaves system.
- Two readers enter system.
- Writer enters system and waits.
- Reader enters system and waits.
- Readers leave system, writer continues.
- Writer leaves system, last reader continues and leaves.

Questions to think about (work it out yourself:

- In case of conflict between readers and writers, who gets priority?
- How would you change the solution so that the priority is switched?

- Can readers or writers get locked out if other threads keep coming in?
- Can you change the solution so that lockout does not occur? (hint: have new readers give priority to any waiting writers and have writers give priority to any waiting readers).
- Can the lines incrementing AR and decrementing WR in release_write_access() be moved to getreadaccess(), just after the OktoRead.down()?
- Can OKToRead ever get greater than 1? What about OKToWrite? Is the first writer to execute Mutex.down() in getwriteaccess() guaranteed to be the first writer to access the data?

# Thread Synchronization with Monitors

Let's revisit the producer-consumer example. The original problem that we were trying to solve was to address two concerns:

- Provide a **locking mechanism** for implementing critical sections
- Provide a way for enforcing **scheduling constraints**, such as a consumer having to wait until there is full slot in the buffer.

With semaphores, we can solve the problem but the solution is not as intuitive as it can be. The use of a counting semaphore works but it is not straightforward to see how to apply it to solve more general scheduling constraints. For example, if we wanted a consumer to proceed only if there was *M* items in the buffer, it is not easy to see how to use a semaphore to model that situation.

*Monitors* provide a simpler solution for enforcing locking and scheduling constraints. The key idea is to provide **separate mechanisms** for the two different types of constraints: locking and scheduling.

## Locking Mechanism:

To implement critical sections, monitors provide methods to create lock objects and to acquire and release locks. For example,

```
Lock mutex;

mutex.lock();
// critical section code
mutex.unlock();
```

Usually, only one lock is used to enforce mutual exclusion on related operations.

## Condition variables (Queues to wait on):

To enforce scheduling constraints, monitors provide methods for threads to wait in a queue when the shared state does not permit progress and for other threads to wake them up when the shared state is appropriate for progress of the waiting threads. Queues on which threads wait are identified by names, called **condition variables**. *Note that condition variables are not really variables that can be assigned values, but objects to wait on.* The only methods allowed on a condition variable c are:

- *c.wait(Lock mutex):* release monitor lock, put thread to sleep on a queue c. When the thread is woken up, via c.signal() or c.broadcast(), it re-acquires monitor lock and then continues in the critical section code.

- *c.signal():* wake up **one** thread waiting on the queue c (typically, FIFO). *If no threads are waiting, c.signal has no effect (no history) – this is very different from semaphore up operation, which has history.*

- *c.broadcast():* wake up **all** threads waiting on the condition variable queue c. *If no threads are waiting, do nothing (no history).*

As a general principle, all these three operations must be called in the critical section protected by the mutex lock.

A typical pattern for a thread that needs to wait for some condition to become true is:

```
Lock mutex;
ConditionVariable cv, othercv;
mutex.lock();     // always the 1st step

// 2nd step below is optional. Only there if there is a  scheduling
// constraint that forces a wait
while (shared state does not satisfy the scheduling constraint)
      cv.wait(mutex);

// assertion: synchronized constraint is satisfied.
// 3rd step: update shared state and state variables if necessary
... code for with the shared state ...

// 4th step: wake up another thread if shared state may permit
// another thread to progress
if (scheduling constraints of others potentially satisfied)
      othercv.signal();   // othercv identifies the constraint queue

mutex.unlock();         // always the last step
```

---

In the above code,

- cv and othercv are two queues on which threads can potentially wait.
- Each thread first acquires a lock before going into the critical section.
- Then, it waits until any scheduling constraints are satisfied by calling the wait primitive on the condition variable.

- The semantics of wait is that the waiting thread *releases* the mutex lock while it is waiting and *reacquires* it sometime after it is woken up and before it continues in the critical section.

- After the thread wakes up from the wait, just to be sure that nothing has gone wrong between the time it wakes up and the time it reenters the critical section, it *re-verifies* that the scheduling constraint is satisfied before continuing. Otherwise, it goes back to waiting.

- Why reverify?

After the scheduling constraint is satisfied, the thread continues in the critical section. First, it makes any updates to the shared state. Then it wakes up any other threads for which the scheduling constraints can be satisfied. Finally, it release the critical section lock.

The above pattern largely works for almost all thread synchronization problems. We illustrate it on two classical problems below:

## ***Producer-consumer Problem:***

Using locks and condition variables, we get a fairly intuitive solution to the producer-consumer problem.

```
Lock mutex;                                  // critical section lock
ConditionVariable OkToProduce, OkToConsume; // queues for waiting
Queue buffer;       // shared buffer
// state variables to enforce scheduling constraints
int numEmpty = N;   // number of empty slots in the buffer
int numFull = 0;    // Number of full slots in the buffer
```

<table>
<tr><th>produceItem(ItemType *item)</th><th>Item *consumeItem()</th></tr>
<tr><td><pre>
mutex.lock();

while (numEmpty == 0)
    OkToProduce.wait(mutex);

buffer.add(item);
numEmpty--;
numFull++;

OkToConsume.signal();
mutex.unlock();
</pre></td><td></td></tr>
</table>

Go through several scenarios to show that the above solution works.

- **Important Question:** Is while loop to reverify the scheduling constraint necessary?

**(NOTE:** There is a difference from the book here!)

There two basic variations of the wait/signal mechanism: Hoare semantics and Mesa semantics. Tanenbaum, like many OS texts, describes only Hoare's semantics in its code, simply because it was proposed first historically. Key difference between the two is:

- In Hoare's semantics: *signalled thread* continues.
    - Signaling thread releases the mutex lock immediately and the signaled thread continues.
    - The signaled thread is **guaranteed** to be the next thread in the monitor. Thus, provided it was woken up when its scheduling constraint was satisfied, the signaled thread does not need to recheck the scheduling constraint.
    - Because of the guarantee, we can often use an "if statement" to check the scheduling constraint rather than the "while" loop.
- In Mesa semantics: *signaller* continues.
    - Signaling thread does *not* release the mutex lock. It continues in the critical section.
    - At some later point, possibly much later, the signaled thread enters the critical section when the lock becomes available. *However, other threads could sneak in into the*

*critical section before the signaled thread* (e.g., someone else buying the JC Penney shirt before you get there). The scheduling constraint could thus become false by the time the signaled thread runs. Thus, *while loop is essential* to recheck scheduling constraint before continuing.

**Almost all real implementations use Mesa semantics** because it is the most practical to use and implement.

**In this course, we will always use Mesa semantics, unless otherwise specified.** ***This is different from the textbook.***

- Convince yourself that the code works when consumers block due to scheduling constraints.

- Does the code work when there are multiple producers and consumers?

- Can signals be replaced by broadcasts?

- Would it be OK to check the scheduling constraint first and then acquire the lock? In other words, switch step 2 and step 1?\

Question: How do  c.wait() and c.signal() compare to semaphore up() and down()?

## Reader and writers problem with monitors:

Procedures which require synchronization between them are declared

as monitor procedures. The entry procedures are:

- `get_read_access()`
- `get_write_access()`

- `release_read_access()`
- `and release_write_access()`

Writer threads do the following in sequence:

- `call get_write_access(): this must block until safe to write`
- `update the database when get_write_access returns`
- `call release_write_access()`

Reader threads do the following in sequence:

- `call get_read_access(): this must block until safe to read`
- `read the database when get_read_access returns`
- `call release_read_access()`

Scheduling Constraints:

- Need a way for reader to block in get_read_access() if a writer is currently active or waiting (we give priority to writers in this solution). To enforce:
    - Use the condition variable *OKToRead*
    - state variables: AW and WW for # of active and waiting writers
- Need a way for writer to block in get_write_access() if a reader or writer is currently active.
    - Use the condition variable *OKToWrite*
    - *state variables: AR* for active readers.

**Question:** Is the database access itself in a critical section?

**Answer:**

As a rule,

- all monitor procedures need to use a mutex lock

- wait, signal, and broadcast need to be called in the critical section

- state variables are shared. Thus, they should only be manipulated in the critical section

See the solution below.

```
Lock mutex;
ConditionVariable OkToRead, OkToWrite;
int AW = 0, WW = 0, AR = 0, WR = 0;

void get_read_access()
{
      mutex.lock();   // step 1: acquire lock
      // step 2:scheduling constraint
      while ((AW+WW) > 0)OKToRead.wait(mutex);
      // step 3: update state variables
      AR++;
      // no step 4: no one to wake up
      // step 5: release lock
      mutex.unlock();


void release_read_access()
{
      mutex.lock(); // step 1
      // step 2: no scheduling constraint

      // step 3: update state variables
      AR--;

      // step 4: wake up anyone necessary
      if (AR==0 && WW>0) OKToWrite.signal();
```

```
        mutex.unlock(); // step 5
}



void get_write_access()
{
        mutex.lock(); // step 1: acquire lock
        // step 2: enforce scheduling constraint
        while ((AW+AR) > 0) {
                WW++;
                OKToWrite.wait(mutex);
                WW--;
        }
        // step 3: update state variables.
        AW++;
        // step 4: no one needs to be woken up
        // step 5: release mutex lock
        mutex.unlock();
}

void release_write_access()
{
        mutex.lock();  // step 1: acquire lock
        // step 2: no scheduling constraint for this method
        // step 3: update state variables. One less active writer
        AW--;
        // step 4: wake up anyone that can potentially progress
        if (WW > 0)
                OKToWrite.signal();
        else
                OKToRead.broadcast(); // multiple readers can progress

        mutex.unlock();   // step 5: release lock
}
```

- Can the while loops be replaced by if conditionals? Note that we are using Mesa semantics.

- Could all of the signals be broadcasts?

- Is WW++ and WW— necessary in get_write_access?

- Exercise for you: how would you modify the release_write_access() code so that waiting readers are given priority over waiting writers? Hint: do you need an additional state variable? Where would that state variable be updated?

# Thread Implementation

We've been talking about

- using threads (concurrency)
- getting them to cooperate
- using sempahores or monitors to enforce mutual exclusion constraints and scheduling constraints

Today, we'll talk about how one actually *builds* them. This material is very relevant to Project 1!

First, note how a process is laid out:

Process:

- body of code
- global data

- one or more stacks (one per thread)

Implementing threads:

Each thread has illusion of its own CPU. Yet on a uniprocessor, all threads share the same physical CPU.

How does this work? We need to be able to do the following operations on threads:

## *Thread state*

Need to save the state of the thread somewhere when it is not running. That structure is called the *Thread control block*. It consists of everything that could be potentially overwritten by another thread:

## *Queues:*

With a single CPU, only one thread can run at a time. When a thread is not running, put it (actually its TCB) on a queue:

- *Ready queue:* contains threads that are runnable.

- *Condition variable queue (one per condition variable):* contains threads that are waiting on a condition variable

- *Lock queues (one per lock):* contains threads that are waiting to acquire a lock

- *I/O or timer wait queues:* contains threads that are waiting for events such as I/O completion or timers to expire. (You do not have to implement these in Project 1)

When a thread is created, it is put on the ready queue. The OS would run threads from the queue, one by one, according to some scheduling policy. One possible policy is to simply run threads in order from the queue.

A thread essentially either runs or get moved to a queue after its state is saved in its TCB. Example:

### *Thread creation (thread forking)*

Let's say that a thread needs to be created that will call foo(arg). We need to allocate a stack and TCB for the thread and make it call foo. When it is has completed foo, we need the thread to destroy itself. The sequence of operations are:

- allocate a stack for the thread
- create and initialize a TCB for the thread with SP pointing to the new stack
- Put the name of the function to be called and its argument on the stack
- set its PC in the TCB to point to a special function, *startThread* so that the thread will call startThread(foo, arg). startThread is responsible for calling foo(arg) and cleaning up thread's state when the call to foo(arg) returns. startThread never returns.
    - Why shouldn't the PC be initialized to the code for foo(arg)?
    - Similar thing happens with the main thread inside a process. The main thread would start with a special

procedure, startProcess(), which calls main(argc, argv), and then marks the thread as DONE so that it can be later deleted by the OS.

- Add the the TCB to the Ready Queue **atomically**. The new thread will run sometime in the future.

**Question:** Does the last operation need to be atomic?

## *Switching Threads*

We switch threads when:

- interrupts, such as timer events, occur:

- thread needing to block on locks or condition variables:
- traps (e.g., divide by 0):
- voluntary yield (thread_yield():
- I/O requests:

Below, we will just look at how switching on voluntary yields can be implemented. On yields, we need the current thread do something like the following:

```
   scheduleAnotherThread() {
      Turn off interrupts
       t =  next thread from the ready queue;
       Current thread puts its TCB at the end of the ready
queue;
      // SWITCH the current thread with thread t so that t
starts running
      SWITCH(t);
      // the new thread now is running. Old thread will
continue from here
      // later when it is switched back.
      Turn on interrupts
    }
```

SWITCH(t) does the following:

- pack up the current thread's state in its TCB
- unpack the state of the new thread t from its TCB. Thus, t automatically continues from where it was saved.

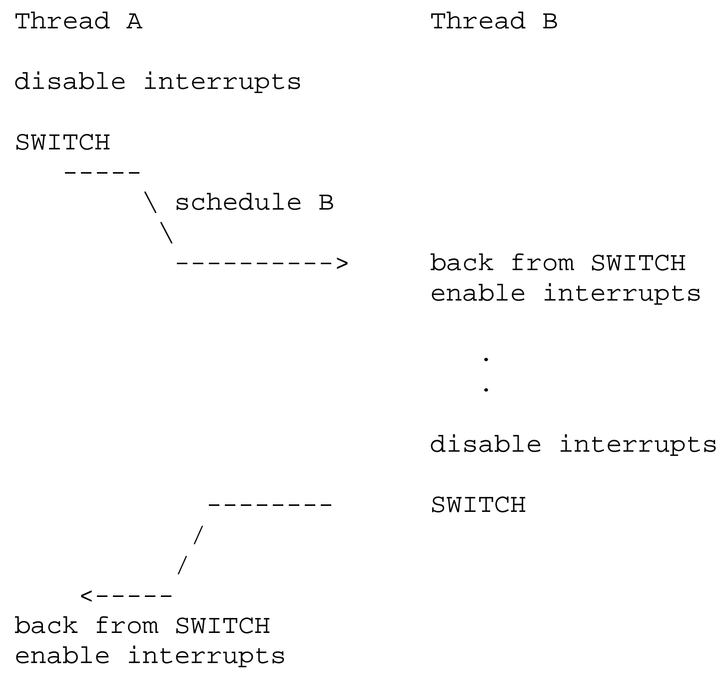
- Packing a thread's state into its TCB:

- We need to save all the thread's registers (tricky, since you need registers to do the saving). Must be done in assembly very carefully without overwriting the registers that you are trying to save.

**Question:** Why does the scheduleAnotherThread() operation need to be made atomic by turning off interrupts? Think about what would happen if there was a context switch to another thread in the middle because of an interrupt?

**Question:** Can interrupts be enabled just before doing the SWITCH?

Thus we do a SWITCH while interrupts are still off. The rule is that all threads always call SWITCH while interrupts are off and turn on the interrupts after the SWITCH. Thus the other thread will turn on interrupts:



If you just look at Thread A's column, you will notice that SWITCH appears like a normal procedure call to A, except that between the time it is called and the time it returns, many other threads can run.

Also notice that interrupts appear to be turned off and turned on correctly in Thread A, despite the fact that thread B ran during the SWITCH. Also, notice that interrupts were turned on correctly while B was running.

**Basic Rule again:** Interrupts must be turned off sometime before the switch is called. They must be turned on sometime after the switch.

## *Thread Termination*

The key difference with thread yielding is that the current thread should not put itself on the ready queue before the SWITCH. Also, it should mark itself as ToBeDestroyed prior to the SWITCH. The thread that runs next should clean up any ToBeDestroyed threads.

- The current thread marks itself as "ToBeDestroyed"

- if another thread is waiting for you to terminate, put it on the ready queue (useful for implementing thread_join() operation).
- Thread puts itself to sleep by calling *sleep():*
    - mark your status BLOCKED
    - find another thread to run from the ready queue.
    - Switch to the other thread

Note: a thread can't destroy itself (e.g., its own stack). Why not?

## *Locks*

**Attempt #1:** One bad but correct solution: Atomic loads/stores as in Peterson's solution. Has busy wait and thus inefficient. Works, but not good and no reason to use it.

**Attempt #2:** run user code with interrupts disabled:

```
lock() {
    disable interrupts;
}

unlock() {
    enable interrupts;
}
```

```
then user program can call lock, unlock around
critical section
```

Many problems:

**Attempt #3:**

main idea: waiting thread gives up processor, goes on the lock queue, until woken up by someone releasing the lock.

```
class Lock {
      ThreadQueue q;
      int value; // BUSY or FREE
}
```

```
Lock::lock() {




}

Lock::unlock() {






}
```

Note that sleep() has similar code to thread_yield(), except that the current thread is not put on the ready queue. sleep() also switches threads by using SWITCH.

Q: why does lock() check value in a while loop? Won't it only get woken up by the unlocker, so it knows lock is free?

Q: Can lock() re-enable interrupts before going to sleep? Why would the solution

break?

1. before putting thread on wait queue?

   scenario that breaks this:

2. after putting thread on wait queue, but before going to sleep?

   scenario that breaks this:

Thus, we must leave interrupts disabled when a thread calls sleep(). The next thread to run has the responsibility of re-enabling interrupts. See picture earlier for SWITCH.

**Question:** With the above solution, are threads guaranteed to get locks in the order that they request the locks?

**Question:** Modify the solution so that threads are guaranteed to get locks in the order that they request them.

## *Condition variables:*

This is something for you to figure out and implement. In implementing the wait(mutex) operation, think very carefully about where interrupts should be disabled and enabled. And recall from the monitor lecture that the wait operation should first cause (1) the mutex lock to be released and (2) the thread to be added to the condition variable queue. Then the thread should sleep (i.e. switch execution to another thread). When the thread is woken up by a signal, it should always reacquire the mutex lock.

## *Locks on Multiprocessors:*

On a multiprocessor system, usually disabling interrupts on one CPU does not disable interrupts on other CPUs. Thus, disabling interrupts to ensure atomicity does not work. Threads on other processors will continue to run.

Fortunately, every modern processor architecture provides some kind of atomic

read-modify-write instruction. *Atomically:*

- read value from memory into register
- write new value

Examples of atomic read-modify-write instruction:

- test&set (most architectures):

    - writes "1" to a memory location ("set"), returns the value that used
    - to be there ("test")
    - note that only 1 thread can get transition from 0->1. Other threads will simply go from 1->1.
- exchange (x86): swap value between register and memory

Here is a solution using atomic test&set instruction:

```
initially, value = 0

lock() {
    while ( test&set(value) == 1) {
    }
}

unlock() {
    value = 0
}
```

A more efficient solution? Don't busy wait for the a thread's entire critical section. Instead, use waiting queues. Replace disabling of interrupts by test&set locks. Look at the the attempt #3 for locks earlier and compare with solution below:

```
lock() {
  while (test&set(guard));      /* this is like "interrupt disable" */

  while (value == BUSY) {
      put on queue of threads waiting for lock
      go to sleep
  }
  value = BUSY;

  guard = 0;              /* this is like "interrupt enable" */
}

unlock() {
  while (test&set(guard)); /* like interrupt disable operation earlier */

  value = FREE
  if anyone on queue of threads waiting for lock {
      take waiting thread off queue
      put on ready queue
  }

  guard = 0;              /* this is like "interrupt enable" */
}
```