Synchronization among Concurrent tasks

In this unit we will consider a very important set of problems in Operating Systems design: synchronization between multiple *overlapping* threads of execution which share data structures in memory. We will see that the issues presented exist in a variety of contexts, although the specifics of how those issues can be solved differs. Specifically, in the UNIX environment, multiple threads of execution exist:

--User level--
• Within a single, conventional (single-threaded) UNIX process, a signal which causes invocation of a handler function can be considered to be a new thread of execution, because the main program code can not control the point at which the signal handler will be invoked. Although the main program's thread is suspended during the execution of the handler, the execution timelines of the handler and the main program overlap and thus can introduce issues of synchronization if both the main program and the handler need to access the same data structure.
• Within a single UNIX process when using a multi-threaded programming environment, e.g. the Posix Threads (pthreads) library. We have not yet formally considered threads programming in this course.
• Among UNIX processes which share a memory region (e.g. through the `mmap` system call). Here the threads of execution are the individual processes. In both this case and the case above, the threads will execute either virtually at the same time (if there is only a single real CPU), or truly in parallel (if there are multiple CPUs or cores)

--Kernel level--
• We will soon see that the Linux kernel is like a giant, multi-threaded program. If there is a single processor, the situation is analogous to a single-threaded UNIX process with handled signals, in that the only synchronization concerns are between what the kernel is doing synchronously on behalf of the process, and what an interrupt handler is doing asynchronously.
• When the Linux kernel is run on a multi-processor system, it is analogous to a multi-threaded UNIX process. There are synchronization issues among these multiple threads of execution, each on an independent processor, as well as with interrupt handlers.

In this unit, we will use the term "thread" and "task" interchangeably and discuss synchronization problems broadly, applying them to all of the above categories.

There are three primary problems in synchronization:
• Maintaining the integrity of data structures when multiple tasks need to update them
• Preventing an erroneous or inconsistent view of data as it is being updated
• Coordinating events between tasks

Shared Update Problem: A Simple Race Condition

Consider an imaginary banking system in which each teller is handled by a separate task. All tasks share a common in-memory data structure which represents the state of all the depositors' accounts. The routine *deposit* updates an account balance:

```
struct account {
          int balance;
          char *name;
          /* etc */
};

deposit(ap,amount)
struct account *ap;
int amount;
{
        ap->balance += amount;
}
```

In this example, the account structure is in memory shared among multiple tasks. On the other hand, the local variable *amount* is private to a particular instance of *deposit*; it is in private memory which is not accessed by any other task (either as an automatic variable maintained on the task's stack or, if the compiler so chooses, held in a register which is part of the task's context).

Now, consider the assembly language instructions implementing the above function (as seen using GCC on 32-bit X86 architecture):

```
deposit:
        pushl   %ebp                            #save ebp register
        movl    %esp, %ebp                      #ebp becomes frame pointer
        movl    8(%ebp), %eax                   #move &ap.balance to eax
        movl    12(%ebp), %ecx                  #move amount to ecx
1       movl    (%eax), %edx                    #FETCH current val to edx
2       addl    %ecx, %edx                      #ADD amount
3       movl    %edx, (%eax)                    #STORE result
        popl    %ebp                            #restore ebp
        ret                                     #return from fn call
```

Note the three instructions with labels 1, 2 and 3, which execute the FETCH, ADD and STORE operations respectively. Abstractly, any computation operation involving memory operands must be performed in these three distinct steps.

Consider an example case where *deposit* is executed simultaneously in two tasks, with *amount*=100, and *balance*=100 before execution begins. The following schedule illustrates proper execution:

Balance	Task A	Task B
100	BEFORE	BEFORE
100	1:fetch balance (=100)	
100	2:add (=200)	
100	3:write balance (200)	
200		1:fetch balance (=200)
200		2:add (=300)
200		3:write balance (300)
300	AFTER	AFTER

There are only two possible sequences of execution which yield the correct answer (300->balance): A1A2A3B1B2B3 or B1B2B3A1A2A3. There are several sequences which yield the wrong answer, e.g.:

Balance	Task A	Task B
100	BEFORE	BEFORE
100	1:fetch balance (=100)	
100	2:add (=200)	
100		1:fetch balance (=100)
100		2:add (=200)
100		3:write (200)
200	3:write (200)	
200	AFTER	AFTER

In this example, between statement 1, which reads the previous value, and statement 3, which writes the new value back, the other task could interlope.

The crux of the problem is that during this window of opportunity, there are multiple copies of the data (the balance). Which one is the "right" one??

Particularly tricky is that the correct operation in this case depends on the essentially random sequence of task scheduling. This is known as a **race condition**. The code could work flawlessly for years on a particular operating system or piece of hardware, and then fail suddenly when the system is updated, the hardware is upgraded, or even when the system load changes resulting in a shift in scheduling order.

Atomicity

The fundamental problem is that the update of the balance is not performed atomically. An operation is deemed to be **atomic** if and only if it can only either run to completion without being interrupted or, if it fails to run, has no effect. Because the update above is not performed atomically, simultaneous or overlapping update requests can interfere with each other.

If we could make the ADD atomic, then there would never be a time where multiple and inconsistent copies of the data would be visible.

(Although this example is somewhat contrived because many processors provide an atomic add instruction, let us continue with it as it is structurally equivalent to more complicated, real-world problems.)

Critical Regions and Mutual Exclusion

In the example above, we can identify the time between and including the execution of step 1 and the completion of step 3 as a **critical region**. Data consistency is vulnerable during a critical region because a private version of a shared object exists or because a shared data structure is in an intermediate state.

Is it necessary to enforce **Mutual Exclusion** in order to guarantee that at most one task is in the critical region at any given time. There are two ways to view mutual exclusion:
• The code-oriented view: mutual exclusion is applied to the critical region between and including steps 1 and 3 of the `deposit` function.
• The data-oriented view: we say that access to a particular `struct account` must be mutually exclusive.

Implementing MUTEX by Disabling Interrupts or Signals

On a uniprocessor system, mutually exclusive access to the critical region can be effected

by preventing any other task from taking control of the cpu during that time. When considering a single-threaded conventional UNIX process, this can be done by blocking all signals upon entry to the critical region, then restoring the previous signal mask on exit:

```
deposit(ap,amount)
struct account *ap;
int amount;
{
 sigset_t oldmask,newmask;
          sigset_fill(&newmask);                 /* Set of all possible signals */
          sigprocmask(SIG_BLOCK,&newmask,&oldmask);       /* Block */
        ap->balance += amount;
          sigprocmask(SIG_SETMASK,&oldmask,NULL);                   /* Restore */
}
```

Likewise, if this example were running within the kernel on a uni-processor machine, analogous protection can be obtained by masking all interrupts on entry to the critical region, then restoring the mask on exit.

However, on a multi-processor system, disabling interrupts only affects the local processor. It would still be possible for `deposit` to be running simultaneously on another processor. At the user-level, blocking signals would not be effective in a multi-threaded environment, or where multiple independent processes share a memory region, because there are other threads of control besides signal handlers.

Spin Lock Mutex

A spin lock is a simple form of mutual exclusion which does not rely on blocking interrupts or signals. The two essential elements of a spin lock are:
• The lock itself, which is simply a variable that has two possible states: locked and unlocked.
• The lock contention mechanism: a tight retry loop.

A task desiring to enter a critical region or access a mutex-protected data structure checks the lock variable. If it is unlocked, it sets the variable to locked and proceeds. If it finds the variable in the locked state, the task executes a tight loop, re-examining the variable until the variable indicates unlocked. On exit from the critical region, the task sets the variable to the unlocked state.

Spin locks only make sense if:
• The critical region is brief so the total number of retries is expected to be small.
• The tasks can not get "stuck" in the critical region thus resulting in an endless spin lock loop.

• The tasks in contention are able to execute in parallel. It would make no sense for a kernel interrupt handler to use a spin lock on single-processor machine, because if the interrupt were to arrive while the kernel was already inside the critical region, the interrupt handler would loop forever waiting for an event which can never come because it requires that the handler return and execution in the main thread resume. We'll see that where spin locks are used inside the Linux kernel, they are protected by a conditional compilation.

As a first-cut approach, we could do the following:

```
struct account {
          int spinlock;                /* 0 is unlocked, NZ is locked */
          int balance;
          char *name;
          /* etc */
};

deposit(ap,amount)
struct account *ap;
int amount;
{
          /* BEGIN CRITICAL REGION */
          while (ap->spinlock!=0)
                    ;
          ap->spinlock=1;
         ap->balance += amount;
          ap->spinlock=0;
          /* END CRITICAL REGION */
}
```

Unfortunately, we have fallen into the same atomicity trap as we had before! Namely, task A could find ap->spinlock clear, but before it has a chance to lock it by setting it to 1, task B could come along and also find 0 in that memory location.

Atomic Instructions

What is required is an instruction which can **atomically test and set.** Such an instruction has the following schematic operation:

```
TAS(int *addr):
          mask interrupts  (this CPU)
          lock memory bus  (all CPUs)
          invalidate other CPU's local caches on addr
          if ((R= *addr)==0)
          {
                    *addr=1
          }
          release memory bus
          unmask interrupts
          return R
```

On a multi-processor system, access to the shared memory pool is coordinated by a **memory arbiter**. When an atomic instruction such as TAS is executed, the local processor locks the memory bus, which instructs the arbiter to prevent any other processor from accessing memory during this critical region. The use of write-back caching introduces additional hardware-level concerns. The hardware must provide a way for this CPU to broadcast a message to all other CPUs causing them to invalidate any cached copy of the memory containing the lock which they might have in their local cache(s).

Processors may have other atomic instructions. For example, in the X86 architecture, many opcodes can be made atomic by prefixing them with the LOCK byte (0xF0). For some of the synchronization methods shown in this unit, these additional atomic instructions may streamline implementation. However, with just one atomic instruction, such as TAS, all other mechanisms can be built up from it.

As an aside, on processors that completely lack any kind of atomic instruction, it is still possible to achieve mutual exclusion through a method known as *Peterson's Algorithm*. This method quickly becomes very awkward when more than two tasks need to be synchronized. This method is now largely of theoretical and historical interest only, since almost all modern processors have atomic instructions (and those that do not typically do not run software that has synchronization issues).

Given that we have an atomic test-and-set operation, our example code can be protected with it:

```
/* DATA-ORIENTED approach */

deposit(ap,amount)
struct account *ap;
int amount;
{
          /* BEGIN CRITICAL REGION */
          while (TAS(&ap->spinlock)!=0)
                    ;
          /* When we get here, we are the only task that could be
           * accessing this struct account */
         ap->balance += amount;
          ap->spinlock=0;
          /* END CRITICAL REGION */
}
```

Code-based or Data-based?

Mutual exclusion could be viewed, as discussed above, as applied to a particular data structure, or applied to a particular region of code. Because the spinlock in the example above is embedded in the `account` data structure, we would tend to think of this as data-oriented locking. Here is the same example, re-written as code-oriented:

```
/* CODE-ORIENTED approach */
```

```
deposit(ap,amount)
struct account *ap;
int amount;
{
  static int deposit_spinlock=0;       /* Protects deposit crit region */
          /* BEGIN CRITICAL REGION */
          while (TAS(&deposit_spinlock)!=0)
                    ;
          /* When we get here, we are the only task holding the deposit_spinlock */
         ap->balance += amount;
          deposit_spinlock=0;
          /* END CRITICAL REGION */
}
```

Unfortunately programmers are often not clear enough about their locking paradigms, and this can cause subtle yet fatal errors when other programmers attempt to write code within the same framework (such as within the kernel). A comment is always in order when locking is used, explaining very clearly what the locking paradigm is and what code and/or data are being locked!

Mutex locking with blocking/yielding

In a multi-processor system, spin locks are reasonably efficient because the task holding the lock will soon release it and only a few cycles will be wasted. However, on a uni-processor system, if task A comes along and finds that the lock is currently held by task B, it means that task B is currently pre-empted, waiting for its turn at the lone CPU. Therefore, things would actually move along faster if task A **blocked** and relinquished the CPU to another runnable task (perhaps task B).

We can also consider, on a multi-processor system, task A holding a mutex for a relatively long time (say a complex data structure change). Task B, running on another process, finds the mutex locked. With spin locks, task B wastes CPU time that could otherwise be allocated to another task C which is ready to run.

Let us define `YIELD` as a function which suspends execution of the current task (and potentially allows another task to execute). When this function returns, the current task has been scheduled again, but this does not imply that the lock is now free. We can define three mutual exclusion locking primitives:

```
mutex_lock(struct mutex *lp)
{
          while (TAS(lp->lock))
                    YIELD(lp);          //includes arranging for wakeup
          return 1;
}

mutex_trylock(struct mutex *lp)
{
          return !TAS(lp->lock);
}
```

```
mutex_unlock(struct mutex *lp)
{
          lp->lock=0;
          WAKEUP(lp);         // wake up any tasks block on mutex *lp
}
```

As we will see in later units, forms of these mutex lock primitives are found both in the kernel code and in the *POSIX Threads* library which provides support for multi-threaded user-level programming. In both cases, the mutex lock is a considerably more complicated data structure as it must store the identifiers of any threads which have blocked on trying to acquire the mutex. When the thread holding the mutex executes `mutex_unlock`, these blocked threads must be notified of this event and run again. This is an example of the **sleep and wakeup** problem. We will have to defer discussion of the mechanics of sleeping and waking up until a subsequent unit.

The deadlock problem

Deadlock occurs when task A is waiting for task B, and vice-versa, or, more generally, there exists any cycle in the wait-for graph depicting the state of all the tasks. Consider the example of tasks A and B which "simultaneously" attempt operations OA and OB, respectively, both requiring locks on resources R1 and R2:

```
OA()
{
         mutex_lock(&R1);
         mutex_lock(&R2);
         /*whatever*/
         mutex_unlock(&R2);
         mutex_unlock(&R1);
}

OB()
{
         mutex_lock(&R2);
         mutex_lock(&R1);
         /*whatever*/
         mutex_unlock(&R1);
         mutex_unlock(&R2);
}
```

Consider this schedule (recall that lock and unlock are atomic primitives)

A	B
lock R1	
	lock R2
	wait for R1
wait for R2	

Deadlock Avoidance

One approach to deadlocks is **deadlock avoidance**. Deadlock can be avoided if locks are **strictly ordered** and all tasks acquire locks in this same order. If any lock can not be obtained, the task must release ALL held locks and try again:

```
obtain_locks()
{
 retry:
        for(i=0;i<N;i++)
        {
                if (!mutex_trylock(&mut[i]))
                {
                        for(j=i;--j>=0;)
                                mutex_unlock(&mut[j]);
                        goto retry;
                }
        }

}
```

Deadlock Detection

Deadlock avoidance isn't always possible for the general case. If the code participating in the potential deadlock isn't written by the same person, or cooperating parties, then it is impossible to enforce ordering. Since the Linux kernel is controlled by one entity, deadlock avoidance is practiced within it for kernel code. But for general-purpose user-level applications, deadlock detection is required:

When an underlying operating system kernel is available to provide the locking primitives, it is possible to perform **deadlock detection**. The operating system can maintain a data structure representing the wait-for dependencies between tasks, and can traverse the graph whenever a lock is requested to see if it would result in deadlock. If so, all tasks involved in the deadlock can be signaled. Deadlock detection is very typically seen in database applications, and in the file-level locking primitives provided by UNIX system calls such as `fcntl`.

Live Lock

A term often used as a corrollary to deadlock is "livelock". When two or more tasks are in conflict with each other such that their states continue to change but eventually return to a previous state without any possibility of ever terminating, that is considered livelock. Another way of defining livelock is an endless loop involving two or more inter-dependent tasks.

Livelock is a complex problem that often perplexes kernel designers and can come in many forms. Resource starvation often triggers livelock. The problem of "thrashing"

seen in Unit 5 is a form of livelock, although it may not meet the strict formal definition, in that two or more memory-starved processes are stuck in a loop of stealing each others' pages.

With respect to mutex locking, there is a formal method to prevent livelock which is similar to the formal method for preventing deadlock:
a) grab locks in a strict order which is consistent for all tasks involved
(b) upon finding a mutex already locked, release all previously acquired locks in the exact reverse of the acquisition order and do not attempt to re-acquire any locks until this is done
(c) the locking primitive must be defined so that when multiple tasks attempt to obtain the lock at the same time, exactly one succeeds. Obviously if more than one gets through, that is a serious error. But if the primitive is defective in that all conflicting tasks fail and nobody gets the lock, that situation can lead to livelock.

Given these requirements, the deadlock avoidance example immediately above is safe against livelock, because we have defined mutex_trylock in such a way that condition (c) is satisfied. In practical applications, livelock and deadlock can creep in because one of these three conditions is violated and that is not immediately obvious from code inspection. We'll see an example soon in the "inconsistent views" section.

```
/* Example of potential livelock */
OA()
{
          for(;;)
          {
                    if (!mutex_trylock(&R1)) continue;
                    if (!mutex_trylock(&R2))
                    {
                              mutex_unlock(&R1);
                              continue;
                    }
          }
         /*whatever*/
         mutex_unlock(&R1);
         mutex_unlock(&R2);
}

OB()
{
          for(;;)
          {
          /* Note that we aren't trying the locks in a consistent order */
          /* This potentially leads to livelock */
                    if (!mutex_trylock(&R2)) continue;
                    if (!mutex_trylock(&R1))
                    {
                              mutex_unlock(&R2);
                              continue;
                    }
```

```
          }
        /*whatever*/
        mutex_unlock(&R2);
        mutex_unlock(&R1);
}
```

Fine vs Coarse Grain Locking, Permissive Locking

If the only object of synchronization was to prevent data corruption, we would always use the biggest, most conservative possible lock. However, another concern is performance. In many real-world cases, a strict mutex lock is too restrictive, in that it prevents one task from getting anything done whatsoever while another task holds a mutex that it needs. We would like to have additional synchronization mechanisms which can be applied at a finer level, and potentially allow more productive simultaneity. In general, we'd like to lock the least amount possible without compromising integrity. Our first example of improving locking granularity will be reader/writer locking.

Read/write locking

When an object is protected by a mutex, this will prevent two tasks from reading the object simultaneously. Since no harm is done by this, it would be desirable to allow it. However, when a task wants to write to the object, that should lock out both other writers and readers, since the object may be in an inconsistent state during the write action.

With read/write locking, any number of tasks may hold a **permissive-** aka **read-only lock** but only one task may hold the **exclusive lock** which allows writing, and only if there are no permissive locks currently held. One possible implementation building on spin locks and the YIELD/WAKEUP mechanism is:

```
struct rwlock {
          int spinlock;      // Protects next two fields
          int readers;
          int writers;
          /* And more stuff for managing blocking/waking */
}

rw_rdlock(struct rwlock *lock)
{
          for(;;)
          {
                    while (TAS(&lock->spinlock)!=0)
                              ;
                    if (lock->writers==0) break;
                    lock->spinlock=0;
                    YIELD();
          }  //spinlock is held on loop exit
```

```
          lock->readers++;
          lock->spinlock=0;
}

rw_wrlock(struct rwlock *lock)
{
          for(;;)
          {
                    while (TAS(&lock->spinlock)!=0)
                              ;
                    if (lock->writers==0 && lock->readers==0) break;
                    lock->spinlock=0;
                    YIELD();
          }  //spinlock is held on loop exit
          lock->writers++;
          lock->spinlock=0;
}


rw_unlock(struct rwlock *lock)
{
          while (TAS(&lock->spinlock))
                    ;
          if (lock->readers>0) lock->readers--;
          lock->writers=0;
          WAKEUP ANY TASKS WAITING ON lock
          lock->spinlock=0;
}
```

The Linux kernel often has "clever" implementations of synchronization mechanisms based on other atomic instructions which are available, especially on the X86 processor family which is its primary target. Within the Linux (32-bit) kernel, reader/write locks are implemented as a 32 bit value. The least significant 24 bits are used as a counter holding the two's-complement of the number of read locks currently held. Bit 24 is the boolean flag and is 1 when all locks are released. Therefore, the value 0x01000000 represents that the read/write lock is unlocked. When the write lock is held, the value is 0x00000000. If, e.g. two tasks hold read locks, the value would be 0x00FFFFFE. (Bits 31-25 are used for bitwise flags by the kernel)

To obtain a read lock, the lock word is atomically decremented. (The X86 architecture has an atomic ADD instruction. Other architectures that lack this must emulate it by using a spin lock to protect a non-atomic add) If this resulted in a zero or negative value, then the lock word must have previously been 0 (write lock already held), therefore the task atomically increments the lock word to undo the previous operation, then waits until the lock word becomes non-zero, then attempts to obtain the lock again.

To obtain a write lock, the value 0x01000000 is atomically subtracted from the lock word. If the result is 0, then it is known that the previous value must have been 0x01000000, i.e. no readers and writers. Furthermore, the current value is now 0, and thus the write lock has been obtained. If the result is non-zero, the task undoes the

operation by atomically adding 0x01000000 and then tries again.

Example of reader/writer locks

In our previous bank account system, we can refine the locking so that write locks are used to make a deposit, but read locks are used to look up a balance.

```
deposit(account,amount)
struct account *account;
{
        rw_wrlock(&account->rwlock);
          account->balance+=amount;
          rw_unlock(&account->rwlock);
        return 1;
}

int viewbalance(struct account *account)
{
 int bal;
          rw_rdlock(&account->rwlock);
          bal=account->balance;
          rw_unlock(&account->rwlock);
          return bal;
}
```

The problem of inconsistent views

The following example illustrates the use of read/write locks, but also introduces a subtle problem:

```
transfer(from,to,amount)        /* Task A /
struct account *from,*to;
{
        if (from==to) return -1;        /*Nonsense operation*/
          for(;;) /* Deadlock avoidance, possible livelock, see text */
          {
                    rw_wrlock(&from->rwlock);
                    if (rw_try_wrlock(&to->rwlock)) break;
                    rw_unlock(&from->rwlock);
          }
        if (from->balance < amount)
        {
                rw_unlock(&to->rwlock);
                rw_unlock(&from->rwlock);
                return -1;
        }
        from->balance-=amount;
        to->balance+=amount;
        rw_unlock(&to->rwlock);
        rw_unlock(&from->rwlock);
        return 1;
}

totalize()              /* Task B */
{
```

```
 int i,tot;
        tot=0;
        for(i=0;i<n_acct;i++)
        {
                rw_rdlock(&accounts[i].rwlock);
                tot+=accounts[i].balance;
                rw_unlock(&accounts[i].rwlock);
        }
        return tot;
}
```

Task B runs periodically to provide a report of the sum total of all accounts in the system. Task A runs from time-to-time to transfer an amount from one account to the other. Clearly task B is read-only with respect to the `balance` field, while task A modifies it. Although the example above protects against any one account having the wrong balance as a result of two instances of task A overlapping, by acquiring writer locks on both accounts before changing them, it does not correctly protect task B against the problem of **inconsistent views**.

Aside: The use of rw_try_wrlock is to address a potential deadlock. Consider what might happen if the blocking rw_wrlock were used on both the to and from account. Numerous deadlocks would be possible. E.g. task A transfers from 100 to 200, B from 200 to 300, and C from 300 to 100. Each task grabs the "from" lock and now A waits for B which waits for C which waits for A: DEADLOCK!

However, the solution presented is not provably correct against LIVELOCK. Consider A transfers from 100 to 200 and B from 200 to 100 and these operations commence simultaneously. A locks 100, B locks 200, A finds 200 locked, B finds 100 locked, A releases 100, B releases 200. A locks 100, B locks 200, and so on....While this is an unlikely situation, to protect against it, we'd either have to introduce a "tie-breaker", e.g. after failing to obtain the second lock, the task waits a random amount of time or an amount of time based on its unique PID; or we could strictly order the locks by checking the memory addresses of the from and to variables, and always locking the one with the lower address first, regardless of whether that is the from or the to.

Suppose task B has just visited account 100 and incremented the grand total by the balance in account 100. Task B releases the reader lock on account 100. Then task A comes along to transfer an amount from account 100 to account 105. Task A will find both accounts unlocked because task B is done looking at #100 and hasn't gotten to #105 yet. The result is that the amount being transferred will be counted twice by task B.

The reason for the failure is that when task B runs, it in fact requires a reader lock on the entire data structure, not just the individual elements of it. Here is a conservative approach to this problem:

```
struct rwlock totalize_lock; // Assume initialized to unlocked

transfer(from,to,amount)        /* Task A /
struct account *from,*to;
```

```
{
        if (from==to) return -1;        /*Nonsense operation*/
          rw_wrlock(&totalize_lock);
          for(;;)
          {
                    rw_wrlock(&from->rwlock);
                    if (rw_try_wrlock(&to->rwlock)) break;
                    rw_unlock(&from->rwlock);
          }
        if (from->balance < amount)
        {
                rw_unlock(&to->rwlock);
                rw_unlock(&from->rwlock);
                return -1;
        }
        from->balance-=amount;
        to->balance+=amount;
        rw_unlock(&to->rwlock);
        rw_unlock(&from->rwlock);
          rw_unlock(&totalize_lock);
        return 1;
}

totalize()              /* Task B */
{
 int i,tot;
          rw_rdlock(&totalize_lock);
        tot=0;
        for(i=0;i<n_acct;i++)
        {
                rw_rdlock(&accounts[i].rwlock);
                tot+=accounts[i].balance;
                rw_unlock(&accounts[i].rwlock);
        }
          rw_unlock(&totalize_lock);
        return tot;
}
```

Unfortunately while this approach prevents the inconsistent view problem, it suffers from terrible performance. While the totalize task is running, all updates to the accounts are locked out. Since the totalizer is a potentially long-running task, this could result in unacceptable delays.

On database systems this problem of inconsistent views is addressed with a Multi-Version Concurrency Control (MVCC) approach, in which each reader is given a private and consistent view of the data until all write activity has completed. While this approach is very interesting, it is completely unsuitable for the synchronization problems found within a kernel, because it relies on a single serializer process (the DBMS daemon process) to arbitrate access and create this consistent view. The kernel does not have the benefit of this external arbiter.

Seqlocks

A possible solution is **optimistic synchronization**, which makes the assumption that most of the time there will be no conflict, and therefore the cost of fixing things when a conflict does occur is more than outweighed by the savings in avoiding excessive locking.

We will use a type of optimistic synchronization which is used in certain places in the Linux kernel, where it is given the name **seqlock**. A seqlock consists of a plain mutex lock and an integer sequence number field, which is initialized to 0. The lock is similar to reader/writer locks, in that different semantics are given to readers vs. writers.

A writer obtains a seqlock by first obtaining the spinlock mutex, and then atomically incrementing the sequence counter by one. To release the seqlock, the counter is incremented once again, and then the mutex is released.

Therefore, if the sequence counter is odd, then the lock is held by a writer. A reader trying to obtain a read lock first examines the sequence counter (without any spin lock or other protection), saving a copy to a local register. If the value read was odd, then the reader waits. If the value is even, the reader is able to enter its critical region and access the data. (This even/odd waiting in and of itself does not solve the entire problem. It merely avoids having the reader make an obviously wasteful incursion into the critical region when it knows that the writer is already there.) At the conclusion of the reader's work, it examines the sequence counter again. If the number has changed, then a writer has interloped, and the reader must try the entire process all over again. This is the other essential part of optimistic synchronization: the ability to reliably detect when the optimism was unfounded, and the ability to harmlessly repeat the operation with hope for a better outcome the next time.

In the previous example, the reader is task B which is calculating the total balance, while task A is a writer. Let us re-write this using seqlocks.

```
/* Seqlock schematic definition */
write_seqlock(struct seqlock *s)
{
          spin_lock(&s->lock);
          atomic_add(&s->count,1);
}

write_sequnlock(struct seqlock *s)
{
          atomic_add(&s->count,1);
          spin_unlock(&s->lock);
}

read_seqbegin(struct seqlock *s)
{
  int a;
          while ((a=s->count)%2)
                    YIELD();
```

```
          return a;
}

read_seqretry(struct seqlock *s,int orig)
{
          return s->count != orig;
}

seqlock_t seqlock;           // Global lock for whole system

transfer(from,to,amount)        /* Task A /
struct account *from,*to;
{
        if (from==to) return -1;        /*Nonsense operation*/
          write_seqlock(&seqlock);
          for(;;) /* Deadlock avoidance */
          {
                    rw_wrlock(&from->rwlock);
                    if (rw_try_wrlock(&to->rwlock)) break;
                    rw_unlock(&from->rwlock);
          }
        if (from->balance < amount)
        {
                rw_unlock(&to->rwlock);
                rw_unlock(&from->rwlock);
                return -1;
        }
        from->balance-=amount;
        to->balance+=amount;
        rw_unlock(&to->rwlock);
        rw_unlock(&from->rwlock);
          write_sequnlock(&seqlock);
        return 1;
}

totalize()              /* Task B */
{
 int i,tot;
 int seq;
        tot=0;
          do {
                    seq=read_seqbegin(&seqlock);          //read first seq#
                    for(i=0;i<n_acct;i++)
                    {
                              rw_rdlock(&accounts[i].rwlock);
                              tot+=accounts[i].balance;
                              rw_unlock(&accounts[i].rwlock);
                    }

          } while (read_seqretry(&seqlock,seq));          //compare to current seq#
        return tot;
}
```

Seqlocks are useful in many places in the Linux kernel. One straightforward example is the `ps` command, which in turn relies on the kernel's `/proc` pseudo-filesystem to return a directory listing in which each node represents a currently running process. This is a reader type of task -- in no way does the process listing modify anything. On the other hand, the fork, exit and wait system calls do change the list of processes, either inserting a new process or removing a dead process. These writer tasks perform the write_seqlock primitives on a seqlock which protects the entire process list, once before they modify it, and again when the modification is complete. The reader task checks the seqlock, assembles a local copy of the list of pids, then checks it again. If the seq# has changed, then the list is not valid, and the reader repeats the attempt and walks the list again. Eventually, unless the system is absurdly busy with fork/exit activity, the reader will assemble the complete pid list without disturbance. This is a far better alternative than having the ps command effectively freeze out any forks or exits.

Task Coordination

Our third major area of synchronization problems concerns coordination among tasks that are dependent on each other to proceed. The classic example is the producer and consumer connected via a FIFO. The consumer can not proceed until the producer puts an element into the FIFO. But if the producer puts elements in faster than they can be consumed, the producer has to wait for the consumer to drain the FIFO. There are several primitives that can be used for task coordination. We will look at two: semaphores and condition variables.

Semaphores

Semaphores are a classic inter-task synchronization mechanism first introduced by Dijkstra in 1965. They are most useful in coordinating producer/consumer tasks or wherever access to a fixed number of resources must be controlled.

A semaphore is a non-negative integer. The integer represents a count of the number of "resources" available and is initialized with that number. When the initial value is 1, the semaphore acts as a mutex lock.

We will define three **atomic** operations:

> "P", which stands for "proberen te verlagen", Dutch for "try to decrease". The P operation checks to see if the semaphore is positive. If it is not, the caller blocks. Otherwise, the semaphore is decremented by 1. This check and decrement is atomic. The P operation is also sometimes called "wait" or "down" or "dec".
>
> "V", which stands for "verhogen", Dutch for "increase". The V operation increments the semaphore by 1. If the semaphore value is now positive, any

sleeping tasks are awakened. V is also called "inc" , "up" or "post".

"TRY", which attempts the P operation, but if the semaphore is not positive, returns immediately with a failure code.

In addition to the counter, a semaphore must contain some way of recording which tasks are blocking on it. The actual blocking and unblocking are not part of the semaphore per se, but must be provided by the operating system.

```
/* A simplistic coordination example using semaphores to manage
 * consumption of a "scarce resource"
*/
struct semaphore okmem;

initialize()
{
          sem_init(&okmem,1024);                /* We have 1024 "slots" */
}

make_something()
{
          sem_wait(&okmem);
          /* allocate the thing */
          return (thething);
}

destroy_something()
{
          sem_inc(&okmem);
}
```

Condition Variables

Condition variables are used to suspend tasks which need to wait for a specific condition (aka event) to occur before proceeding. When another task generates the event, it wakes up any tasks which are sleeping on the condition variable. An underlying operating system must provide sleep and wakeup primitives.

The condition variable facilities used in the example below are found in the POSIX threads library and are available under Linux, Solaris and many other operating systems. The Solaris kernel relies heavily on condition variables. The Linux kernel uses a very similar facility called "wait queues", which will be discussed in a subsequent unit, and semaphores, which are discussed above.

Condition Variable Operations

A condition variable is always used in conjunction with a mutex lock. The following condition variable operations are provided:

• `pthread_cond_wait`: Block the calling task on the condition variable (first argument), which represents a specific condition or event the task wishes to wait for. A mutex is supplied as the second argument. The identity of the task calling `pthread_cond_wait` is stored in the condition variable data structure, the supplied mutex is released and the calling task is put to sleep atomically. The task sleeps until it is awakened by some other task calling `pthread_cond_broadcast` or `pthread_cond_signal`. Before returning control to the caller, `pthread_cond_wait` re-acquires the mutex. Schematically, we can express this primitive as:

```
pthread_cond_wait(pthread_cond_t *cp,pthread_mutex_t *mut)
{
        ATOMICALLY {
                note within *cp that we are interested
                pthread_mutex_unlock(mut)
                go to sleep
        }
        /* We have been awakened */
        pthread_mutex_lock(mut);
}
```

• `pthread_cond_signal`: Release at most one task blocked on the supplied condition variable. If more than one task is blocked on the condition variable, one is picked (the POSIX definition does not define which one but many implementations pick the longest-waiting.) If no tasks are blocked, this call has no effect, and is not "remembered". Should be called under the protection of the mutex for the condition variable, as discussed below.

• `pthread_cond_broadcast`: Release all tasks blocked on the supplied condition variable. If there are no blocked tasks, this call has no effect and is not "remembered". Should be called under the protection of the mutex for the condition variable, as discussed below.

Producer/Consumer problem using condition variables

```
volatile struct fifo {
          char buf[BSIZE];
          int next_write,next_read;
          int item_count;
          pthread_cond_t full,empty;   /* Condition variables */
          pthread_mutex_t mutex;                /* Common mutex for fifo */
} fifo;

initialize(struct fifo *fifo)
{
          fifo->mutex=PTHREAD_MUTEX_INITIALIZER;
          fifo->full=PTHREAD_COND_INITIALIZER;
          fifo->empty=PTHREAD_COND_INITIALIZER;
          fifo->next_write=0;
          fifo->next_read=0;
          fifo->item_count=0;
}

producer(struct fifo *fifo,char c)
{
          pthread_mutex_lock(&fifo->mutex);
          while (fifo->item_count>=BSIZE)                  /* Full */
                    pthread_cond_wait(&fifo->full,&fifo->mutex);
          fifo->buf[fifo->next_write++]=c;
          fifo->next_write%=BSIZE;
          fifo->item_count++;
          pthread_cond_signal(&fifo->empty);
          pthread_mutex_unlock(&fifo->mutex);
}


consumer(struct fifo *fifo)
{
 char c;
          pthread_mutex_lock(&fifo->mutex);
          while (fifo->item_count<=0)            /* Empty */
                    pthread_cond_wait(&fifo->empty,&fifo->mutex);
          c=fifo->buf[fifo->next_read++];
          fifo->next_read%=BSIZE;
          fifo->item_count--;
          pthread_cond_signal(&fifo->full);
          pthread_mutex_unlock(&fifo->mutex);
          return c;
}
```

When the condition variable is a simple counting expression, such as the FIFO example above, then condition variables and semaphores are virtually equivalent. But condition variables can also control more complicated boolean expressions which do not obey this count-up/count-down paradigm.

Some Golden Rules of Condition Variables

• **The condition variable is not the condition.** The condition is some boolean expression which is tested separately.
• **The condition must be re-tested.** Return from `pthread_cond_wait` does not guarantee that the condition is now true, because it is **not atomic** with respect to the condition: between the time that the task wakes up and the time that it re-acquires the mutex within `pthread_cond_wait`, another task might come along, acquire the mutex, and perform an operation which causes the condition to become false again. In the producer/consumer example, we see that the empty or full condition is tested with a while loop.
• **The mutex must be held while the condition is being tested**. The condition test itself is not necessarily atomic (e.g. the boolean expression is made up of && or || clauses). Furthermore, testing the condition and calling `pthread_cond_wait` are not inhererently atomic. They must be made atomic through the use of the mutex. Consider the FIFO example above without a mutex. Between the time that, e.g., the consumer finds the fifo empty in the while() test and the time it puts itself to sleep with pthread_cond_wait a producer could come along, write a character (rendering the empty condition false) and call pthread_cond_signal. Since the consumer has not reached cond_wait yet, the signal is lost and the consumer may sleep forever. This is known as the **lost wakeup problem**. By protecting access with a mutex, the test and the sleep effectively become atomic, preventing a lost wakeup.
• **Use** `pthread_cond_signal` , only when at most one awakened thread would be able to proceed. Otherwise use `pthread_cond_broadcast` to make sure all threads which could possibly proceed are woken up.

• **Hold the mutex when** `pthread_cond_signal` **or** `cond_broadcast` **is called**. If not, another thread may be testing the condition just as the wakeup arrives and would therefore miss the wakeup and potentially wait forever.