Concurrency Control Primitive

Cover concurrency control for single object, excluding transactions. [Lecture 1-4]

Reference

- UCSD CS
  - L6 concurrency control
  - L7 semaphores and condition variables
  - L8 monitor and condition variables
- Cambridge
  - Concurrency 2022/23 Learners' Guide
  - Textbook Operating Systems-Concurrent and Distributed Software Design

Key Ideas

Safety

- Mutual Exclusion in Critical Sessions
  - Only one process can be in the critical section at a time

Progress

- No process is forced to wait for an available resource
  - wakeup / conditional synchronization
- Bounded Waiting
  - No process can wait forever for a resource (Deadlock) [Lecture 5]

Hardware atomic operation

Computer Architecture: Memory Consistency

- Uninterruptible sequences of operations that appear to all occur as one.
  - Read-Modify-Write
- Usage: Locks, Thread Barriers
- Read-and-Set, which can be fully run in user space (implemented as a machine instruction)

Compare-and-Swap

- expected value = free, new value = occupied.
- allows the atomic compare to check if the state was free
- and setting the new value to a memory location, return its old value.

The semantics of the instruction are below. Please remember that it is atomic and **not** interruptible.

```
C&S (<mem loc>, <expected value>, <new value>)
{
    if (<mem loc> == <expected value>)
    {
        <mem loc> = <new value>;
        // loop when store failed
        return 0;
    }
    else // owned
    {
        return 1;    // failed
    }
}
```

Spin lock with CAS

The pseudo-code below illustration the creation of a mutex (spin lock) using compare-and-swap:

- A process busy-waits until the compare-and-swap succeeds, at which time it can move into the critical section.

```
/* Before entering critical section */
Acquire_mutex(<mutex>)
{
    while (CS (<mutex>, 1, 0))
    ;
}
```

- When done, it can mark the critical section as available by setting the mutex's value back to *the expected value 1.*

```
/* After exiting critical section */
Release_mutex(<mutex>)
{
    <mutex> =  1;
}
```

Test-and-Set

Relationship: test-and-set is just *a special case* of compare-and-swap:

```
T&S(x) == C&S (x, 0, 1)
```

- 0 = free ; 1 = occupied.
- allows the atomic testing if the state was free
- and setting the value of 1 to a memory location, return its old value.
- if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process's test-and-set is finished.

The semantics of the instruction are below. Please remember that it is atomic and **not** interruptible.

```
T&S (<mem loc>)
{
    if (<memloc> == 0)
    {
        <mem loc> = 1;
        // loop when store failed
        return 0;
    }
    else  // owned
    {
      return 1;     // failed
    }
}
```

Spin lock with TAS

- A process busy-waits until the test-and-set succeeds, at which time it can move into the critical section.

```
/* Before entering critical section */
Acquire_mutex(<mutex>)
{
    while(TS(<mutex>))
}
```

- When done, it can mark the critical section as available by setting the mutex's value back to 0

```
/* After exiting critical section */
Release_mutex(<mutex>)
{
    <mutex> = 0;
}
```

More

- Fetch-and-Add/DECrement Condition
- LL/SC

Comparison:

- Old atomic instructions, such as Test-and-Set or Compare-and-Swap require the memory system busses to be locked for the duration, likewise precluding concurrency.
    - Not good for multi-core, shared-memory designs.
- LL/SC is now preferred.
    - No global lock down.
    - Memory can be partitioned into one or more disjoint logical banks and associated with each bank is a record of processor/core ID that last issued LL. SC only succeeds if no other core has intervened on that bank in the meantime (precise definition of intervened is hardware-specific but always includes issuing an LL instruction). This is optimistic concurrency control (see later), and, as always, requires clients to re-try on failure. Failure should be rare.

FSM

Thread state

- start-up, live and deadlocked/livelocked/bad state sets

Spin Lock as mutex

Spin lock is a **binary counter**, supported by hardware CAS, LL/SC.

- The counter supports two operations (atomic by hardware)
- Hardware operation can be fully run in user space (implemented as a machine instruction)
- 1: Consumer P(x) / test / Lock / wait, named after Proberen (Dutch).
    - feature: busy wait, easy but *inefficient* if contention is high and/or the critical section is large. Improved by binary semaphore.

```
/* proberen - test / Lock */
P(sem)
{
    while (sem <= 0){
      continue;        /* spin, i.e. busy wait  */
    }
    sem = sem - 1;
    // loop when store failed
}
/* Hardware Fetch-and-DECrement Condition*/
```

- 2: Producer V(x) / increment / Unlock, named after Verhogen (Dutch).

```
/* verhogen - to increment / Unlock */
V(sem)
{
    sem = sem + 1;
}
```

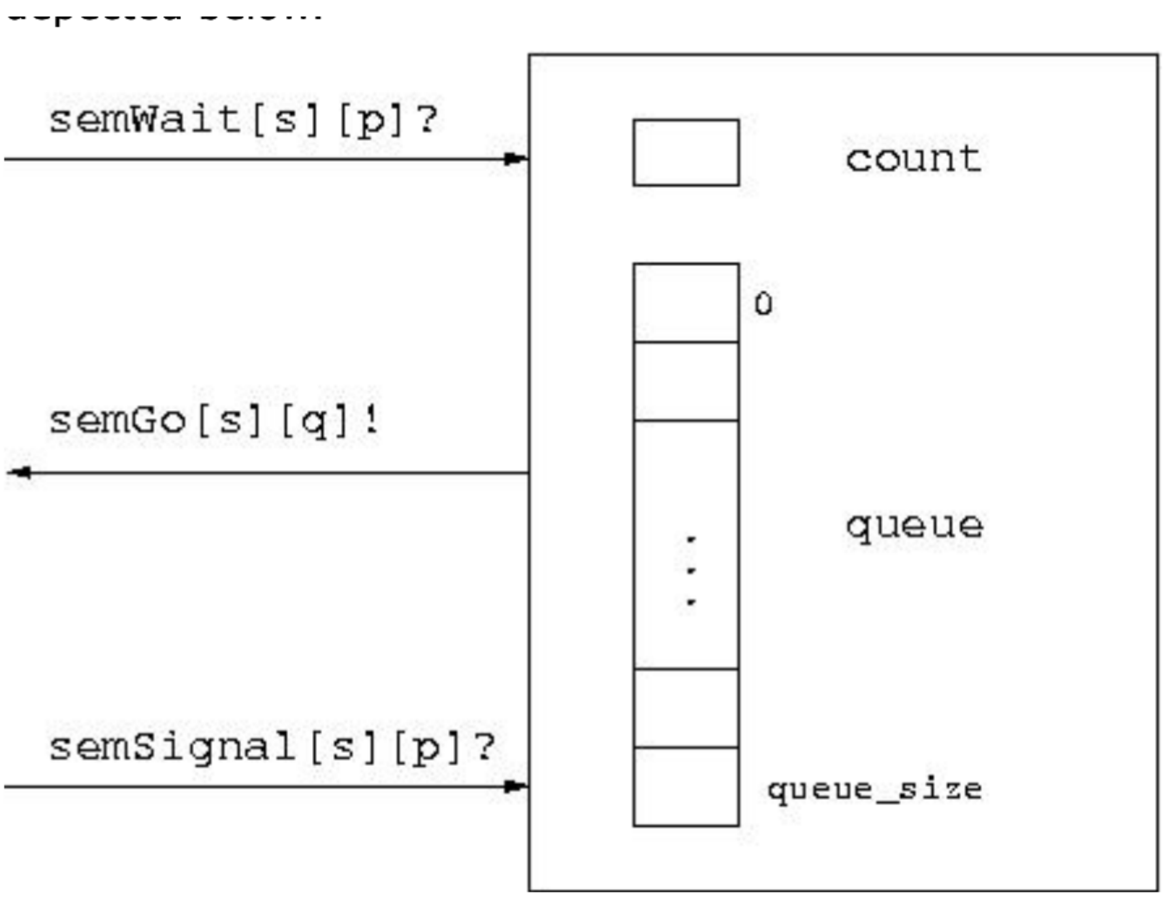
Condition Critial Region

CCR

- Implicit Spin lock

```
Region X await <cond> do Y
```

Semaphore



Semaphore is a counter + **a mutex + a wait queue, with system call.**

- a **counter** representing the resource instance number.
- **mutex** for composite operations (add and signal) in the critical sections within the semaphores themselves, again growing a smaller guarantee into a larger atomic body.
- progress is possible
  - **block()** instead of spin [- in the code]

    - **wake up** / conditional synchronize the wait queue members. [+ in the code]
- The counter supports two operations (atomic by hardware)
    - Consumer P(x) / test / down / wait, named after Proberen (Dutch).

```
/* proberen - test / Lock */
P (csem) {
    while (1)  {
      Acquire_mutex (csem.mutex);
      if (csem.value <= 0) {
          +  insert_queue (getpid(), csem.queue);
          Release_mutex_and_block (csem.mutex);
          /* atomic: otherwise lost wake-up */
          - continue;     /* No more spin  */
      }
      else {
          csem.value = csem.value - 1;
          Release_mutex (csem.mutex);
          break;
      }
    }
}
```

- Producer V(x) / increment / up / signal, named after Verhogen (Dutch).

```
V (csem)
{
    Acquire_mutex (csem.mutex);
    csem.value = csem.value + 1;
    + dequeue_and_wakeup (csem.queue)
    /* via IPI system call */

    Release_mutex (csem.mutex);
}
```

Lost wakeup

P()ing process must atomically become not runnable and release the mutex, because of the risk of a *lost wakeup*.

```
Release_mutex_and_block(csem.mutex);
/* atomic: otherwise lost wake-up */
```

When process P1 releasing the mutex, if an **interrupt** (e.g., *context-switch*) occurs between the `release_mutex(csem.mutex)` and the `block/sleep()`, it would be possible for another process P2 to perform a V() operation, before first process P1 asleep.

When P2 attempts to dequeue_and_wakeup() the first process P1. Unfortunately, the first process may **not** yet asleep, so it missed the wake-up. When P1 again is scheduled to run, it immediately goes to sleep with **no one** left to wake it up.

Solution

- **System call** and kernel puts the process to sleep free of interruptions.

```
sleep(mutex); //syscall
```

- OS generally provides support in the form of a sleep() *system call* that takes the mutex as a parameter. The kernel can then release the mutex and put the process to sleep in an environment free of interruptions (or otherwise protected).

Note:

- This atomic "unlock and wait" is the primary purpose of condition variables and the reason they must always be associated with a mutex and a predicate.

Usage

Counting Semaphore

```
public static Semaphore semOne = new Semaphore(N);  // N > 1
```

By initialising a semaphore to N > 1, we provide part of the management structure needed to prevent a number of jobs/people/masters overloading a pool of N servers. MP-MC (Multiple Producer-Multiple Consumer).

Binary / Boolean Semaphore

```
public static Semaphore semOne = new Semaphore(1);
```

By initialising the semaphore to one, just one thread could be in a mutual exclusion region.

- There is only one resource.
- Boolean semaphores may only have a value of 0 or 1.

Condition Synchronisation

```
public static Semaphore semOne = new Semaphore(0);
```

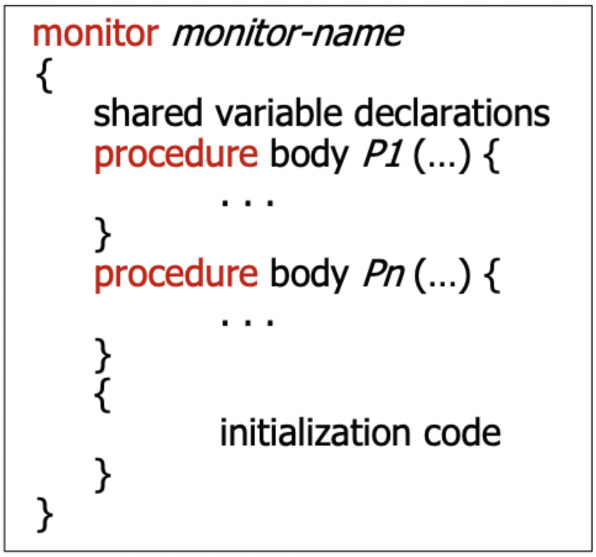
By initialising a semaphore to zero, only threads of the *creating process* can use the semaphore, no other slave could call `acquire()`.

It provides a condition synchronisation mechanism, where a slave/server thread can held queued, waiting for work to arrive from a master thread.

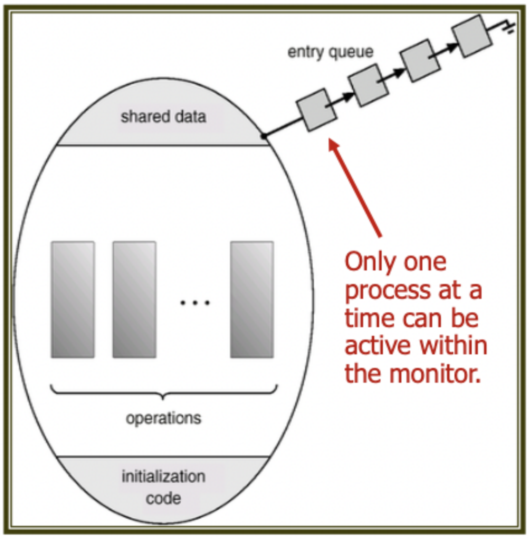
Keep Invariance

Protect the temporary state of data structures when mutated from being observed by others.

Monitor



```
monitor monitor-name
{
     shared variable declarations
     procedure body P1 (...) {
               . . .
     }
     procedure body Pn (...) {
               . . .
     }
     {
               initialization code
     }
}
```


entry queue
shared data
...
operations
initialization code
Only one process at a time can be active within the monitor.

- Explicit **conditional variable** is essentially a wait-queue for threads waiting to enter the monitor
    - with implicit mutual exclusion in the monitor.
- condition (data structure) possible implementation: a double-linked list to use as a queue. It also contains a semaphore to protect operations on this queue. This semaphore should be a spin-lock since it will only be held for very short periods of time.

```
struct condition {
    proc next;  /* doubly linked list implementation of */
    proc prev;  /* queue for blocked threads */
    mutex mx;   /* implicit mutex protects queue */
};
```

- supports three operations
    - wait
        - add calling thread to the queue and put it to sleep
    - signal
        - remove a thread form the queue and wake it up
    - broadcast
        - remove and wake-up all threads on the queue
- Two semantics
    - Should the caller of signal wait or continue?

Usage

- Declaration in monitor

```
condition notfull = True, notempty = False;
```

- single implicit mutex

Only one thread is allowed in the monitor.

```
GetLock (condition cv)
{
    /*if (LOCKED-CONDITION) - only safe in signal and wait */
    while (LOCKED-CONDITION)     /* safe in both semantics */
      wait (cv);
}
```

In the case of a *sharable* resource, a *broadcast* can be sent to wake up all sleeping threads.

```
ReleaseLock (condition cv)
{
    if (UNLOCKED-CONDITION)
      signal (cv);
}
```

- Extension: multiple explicit mutexes, each for sharable variables.

A thread tests to see if the resource is available.

- If it is available, it uses it.
- Otherwise, it adds itself to the queue of threads waiting for the resource.
- Notes that both the mutex `mx` and the condition variable `cv` are passed into the wait function. (whereas, mutex could be omitted for single mutex.)

```
GetLock (condition cv, mutex mx = spin_lock ss)
{
    mutex_acquire (mx);
    /*if (LOCKED)      only safe in signal and wait */
    while (LOCKED)  /* safe in both semantics */
      wait (cv, mx);

    lock=LOCKED;
    mutex_release (mx);
}
```

Note:

If you examine the implementation of wait, you will find that the wait function atomically releases the mutex and puts the thread to sleep. After the thread is signalled and wakes up, it reacquires the resource. This is to prevent a lost wake-up. This situation is discussed in the section describing the implementation of condition variables.

In the case of a *sharable* resource, a *broadcast* can be sent to wake up all sleeping threads. Otherwise only to no more than threads.

```
ReleaseLock (condition cv, mutex mx)
{
    mutex_acquire (mx);
      lock = UNLOCKED;
      signal (cv);
    mutex_release (mx);
}
```

Extension: Condition operations implementation

- wait()

The wait() operation adds a thread to the list and then puts it to sleep. The mutex that protects the critical section in the calling function is passed as a parameter to wait(). This allows wait to atomically release the mutex and put the process to sleep.

If this operation is not atomic and a context switch occurs after the release_mutex (mx) and before the thread goes to sleep, it is possible that a process will signal before the process goes to sleep. When the waiting() process is restored to execution, it will enter the sleep queue, but the message to wake it up will be forever gone.

```
void wait (condition *cv, mutex *mx)
{
    mutex_acquire(c->listLock);  /* protect the queue */
    enqueue (&c->next, &c->prev, thr_self()); /* enqueue */
    mutex_release (c->listLock);   /* we have finished with the list

    /* The suspend and release_mutex() operation should be atomic */
    /* otherwise we will meet a lost wakeup. */
    release_mutex (mx);
    thr_suspend (self);  /* Sleep 'til someone wakes us */

    mutex_acquire (mx); /* Woke up -- our turn, get resource lock */

    return;
}
```

- signal()

The signal() operation gets the next thread from the queue and wakes it up. If the queue is empty, it does nothing.

```
void signal (condition *c)
{
    thread_id tid;

    mutex_acquire (c->listlock); /* protect the queue */
    tid = dequeue(&c->next, &c->prev);
    mutex_release (listLock);

    if (tid>0)
      thr_continue (tid);

    return;
}
```

- broadcast()

The broadcast operation wakes up every thread waiting for a particular resource. This generally makes sense only with *sharable* resources. Perhaps a writer just completed so all of the readers can be awakened.

```
void broadcast (condition *c)
{
    thread_id tid;

    mutex_acquire (c->listLock); /* protect the queue */
    while (&c->next) /* queue is not empty */
    {
      tid = dequeue(&c->next, &c->prev); /* wake one */
      thr_continue (tid); /* Make it runnable */
    }
    mutex_release (c->listLock); /* done with the queue */
}
```

Read-Write Lock

The MRSW paradigm

- allows multiple threads to have read-only access on a shared resource
    - `read` semaphore among readers and a *counter* to check the readers inside
    - when the first reader comes in, check not `write` [invariant]
    - when the last reader exits, signal `write` [invariant]
- or a single thread to have read-and-write access
    - `write` semaphore among writers
- Fairer by another `turn` semaphore
    - acquired by writer when entering
    - writer could enter after all current readers exit
    - block further readers until writer finishes.
- Note: real-world can provide a further mechanism *upgrade-to-write*

    - a thread holding a read lock can convert this to an exclusive write lock by waiting for all other readers to complete. (2PL, y2022)
    - Note: holding a write lock also allows reading.
- implement using semaphore primitive or as a monitor. (Exercise Sheet 0)

Transaction and ACID

Concurrency control for multiple objects. [Lecture 6-8]

Reference:

CMU15-445 Database Management System

Isolation

	(Strict) 2PL	**(Strict) TSO**	**OCC**
Key	Lock /**MR**SW	TS start	TS validation
Deadlock	*may deadlock*	No	No
Distributed	x	*distributed*	x (usually)

No cascading abort
& recovery (undo)
	if unlock
	until commit
		if delay
		until commit
			strict ✓

Decider	the lock manager	each object	the validator
Degree of concurrency	low	high	higher
Optimism	x pessimism	✓	✓**rare conflict**

Durability

- Write-ahead log
- Checkpoints
- Recovery and rollback