Topic 9: Synchronization via Semaphores

- Requirements for a mutual exclusion mechanism:
  - Must allow only one process into a critical section at a time.
  - If several requests at once, must allow one and only one process to proceed.
  - Processes must be able to go on vacation outside the critical section.
  - Must handle an arbitrary number of processes.
- Desirable properties for a mutual exclusion mechanism:
  - Fair: if several processes waiting, let each in eventually.
  - Efficient: do not 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).
- Desirable properties of 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 (there are a few exceptions to this…).
  - Do not spend large amounts of time in critical section.

- <u>Semaphore</u>
  - Synchronization variable that takes on non-negative integer values.
  - Invented by E. Dijkstra in the mid-60's.
  - Patrick's train story here…

- `P( semaphore )`:
  - Atomic operation that waits for semaphore to become positive, then decrements it by 1.
  - "Proberen" in Dutch ("try")

- `V( semaphore )`:
  - Atomic operation that increments semaphore by 1.
  - "Verhogen" in Dutch ("increase")

- Semaphores are simple and elegant. They allow the solution of many interesting problems.
- They can do a lot more than just mutual exclusion.

- Too much milk problem with semaphores:

```
Processes A & B

P( OKToBuyMilk );
if ( noMilk ) {
    BuyMilk();
}
V( OKToBuyMilk );
```

  - Notes: **`OKToBuyMilk`** must initially be set to 1.
    - What happens if it is set to zero? set to two? etc.
- *Binary Semaphores*
  - Have only two possible values: 0 and 1.
  - Used for mutual exclusion.
- *Counting Semaphores*
  - Have any non-negative value (>= 0).
  - Used for scheduling
    - To wait for some event to happen.
    - To wait for some resource to become available.

- Semaphore properties:
  - + Machine independent.
  - + Simple (to use).
  - + Works with many processes (not just two).
  - + Can have many critical sections in the code, each with its own semaphore.
  - + Can acquire many resources simultaneously (multiple P's).

```
/* I want three of these to go, please… */
P( pizza );
P( pizza );
P( pizza );
… using the resources to make 3 pizzas …
V( pizza );
V( pizza );
V( pizza );
```

    - What happens if only 2 pizza resources are available?
  - - Not provided by hardware.

Semaphores are used in two different ways:

- **Mutual Exclusion**:
  - Ensure that only one process is accessing shared information at a time.
  - If there are separate groups of data that can be accessed independently, there may be separate semaphores, one for each group of data.
    - These are also binary semaphores.

- **Scheduling**:
  - To permit processes to wait for certain things to happen.
  - If there are different groups of processes waiting for different things to happen, there will usually be a different semaphore for each group. These semaphores are not necessarily binary semaphores.
  - <u>Resource allocation</u> is an important form of scheduling.
    - If there are N items of a resource, then set an associated semaphore to N initially.
    - P to acquire the resource.
    - V to release the resource.
    - Processes block (on P) when the resource runs out and wait for more to be released (V).

Semaphore Example: Producer — Consumer:

- Suppose one process is creating information that is going to be used by another process.
  - E.g., one process is reading a program from the disk, and another process will compile the program.
  - E.g., one process is building a list of processes, and another process is printing only those processes owned by you.
  - E.g., Unix pipes
- Processes should not have to operate in perfect lock-step: producer should be able to get ahead of consumer.
- Producer: creates copies of a resource.
- Consumer: uses up (destroys) copies of a resource.
- Buffers: used to hold information after producer has created it, but before consumer has used it.
- Synchronization: keeping producer ahead of consumer.
- Define constraints (definition of what is "correct").
  - Consumer: if all buffers are empty, must wait for producer to fill buffer(s). Scheduling.
  - Producer: if all buffers are full, must wait for consumer to empty buffer(s). Scheduling.
  - Only one process may manipulate buffer pool at a time. Mutual exclusion.
  - A separate semaphore is used for each constraint (= 3 semaphores).

- Initialization:
  - Put all buffers in a pool of empties.
  - Initialize semaphores (specify the initial value of the semaphore when creating it):

```
semaphore empties = sem_create( NUM_BUFFERS );
semaphore fulls = sem_create( 0 );
semaphore mutex = sem_create( 1 ); /* 'mutex' = mutual exclusion */
```

- The solution:

Producer	**Consumer**
`P( empties );`	`P( fulls );`
`P( mutex );`	`P( mutex );`
`<get empty buffer from pool of empties>`	`<get full buffer from pool of fulls>`
`V( mutex );`	`V( mutex );`
`<put data in buffer>`	`<consume data in buffer>`
`P( mutex );`	`P( mutex );`
`<add full buffer to pool of fulls>`	`<add empty buffer to pool of empties>`
`V( mutex );`	`V( mutex );`
`V( fulls );`	`V( empties );`

- Questions:
  - Why does producer **P(empties)** but **`V( fulls )`**?
  - Why are data in the buffer accessed outside the critical section?
  - Is the order of first two P's important in each case?
  - Is the order of the last two V's important in each case?
  - Could we have separate semaphores for each pool?
  - How would this be extended to have two consumers?
  - How about two producers?

Semaphore Example: Readers — Writers:

- Shared database with readers and writers.
- It is safe to have several readers access the database simultaneously.
- A writer must have exclusive access.
  - No other writers.
  - No other readers.
- Invariant:

```
(ActiveWriters == 1 && ActiveReaders == 0) || (ActiveWriters == 0 && ActiveReaders >= 0)
```

- Use semaphores to enforce these policies.
  - Note: Writers are actually readers too.
- Constraints:
  - Writers can only proceed if there are no active readers or writers. Use semaphore **`OKToWrite`**.
  - Readers can only proceed if there are no active writers.
    - Implies that readers get priority.
  - Need a shared variable to keep track of who is reading.
    - Must ensure that only one process manipulates shared variable at once — need a mutex semaphore.

- State variables
  - `ActiveReaders` = number of active readers.
  - `WaitingReaders` = number of waiting readers.
  - `ActiveWriters` = number of active writers.
  - `WaitingWriters` = number of waiting writers.
  - `ActiveWriters` is always 0 or 1.
  - `ActiveReaders` and `ActiveWriters` may not both be non-zero at the same time.
- Initialization:

```
semaphore OKToRead = create_sem(0);
semaphore OKToWrite = create_sem(0);
semaphore Mutex = create_sem(1);
int ActiveReaders, WaitingReaders, ActiveWriters, WaitingWriters;
ActiveReaders = WaitingReaders = ActiveWriters = WaitingWriters = 0;
```

- Scheduling
  - Writers-preference.
  - Note: readers-preference and no-preference solutions are also possible.

Reader Processes

```
P( Mutex );
if ( (ActiveWriters == 0) &&
     (WaitingWriters == 0) ) {
   V( OKToRead );
   ActiveReaders++;
}
else
   WaitingReaders++;
V( Mutex );


P( OKToRead );
   <read the data…>

P( Mutex );
ActiveReaders--;
if ( (ActiveReaders == 0) &&
     (WaitingWriters > 0) ) {
   /* wake up a writer */
   V( OkToWrite );
   ActiveWriters++;
   WaitingWriters--;
}
V( Mutex );
```

Writer Processes

```
P( Mutex );
if ( (ActiveWriters == 0) &&
     (ActiveReaders == 0) &&
     (WaitingWriters == 0) ) {
   V( OKToWrite );
   ActiveWriters++;
}
else
   WaitingWriters++;
V( Mutex );

P( OKToWrite );
   <write data…>

P( Mutex );
ActiveWriters--;
if ( WaitingWriters > 0 ) {
   V( OKToWrite ); /* wake up a writer */
   ActiveWriters++;
   WaitingWriters--;
}
else { /* wake up all the waiting readers */
   while ( WaitingReaders > 0 ) {
      V( OKToRead );
      ActiveReaders++;
      WaitingReaders--;
   }
}
V( Mutex );
```

- Consider several examples:
  - Reader enters and leaves system.
  - Writer enters and leaves system.
  - Two readers enter system.
  - Writer enters system (and has to wait).
  - Reader enters system (wait? what? where?).
  - Two readers leave system — writer does what?
- Questions:
  - In case of conflict between readers and writers, who gets priority? What lines of the code help insure this?
  - Is the **`WaitingWriters`** necessary in the writer's first **`if`**?
  - Can **`OKToRead`** ever be greater than 1?
  - Can **`OKToWrite`** ever be greater than 1?
  - Is the first writer to execute **`P( Mutex )`** guaranteed to be the first writer to access the data?

Semaphore Implementation.

- No existing hardware implements semaphores directly.
  - All involve some sort of CPU scheduling; it is not clear that scheduling decisions should be made in hardware anyway.
- Must be built up in software using a lower-level synchronization primitive provided by hardware.
- Need a simple way of doing mutual exclusion in order to implement P's and V's.
- Could use atomic reads and writes, as in the too-much-milk problem. Clumsy!
- Uniprocessor solution: disable interrupts.

```
typedef struct {
   int count;
} Semaphore;
```

```
P( Semaphore *sem ) {
   while (1) {
      interruptsDisable();
      if ( sem->count > 0) {
         sem->count--;
         break;
      }
      interruptsEnable();
   }
   interruptsEnable();
} /* P */
```

```
V( Semaphore *sem ) {
   interruptsDisable();
   sem->count++;
   interruptsEnable();
} /* V */
```

- Step 1:
  - When P fails, put process to sleep.
  - On V just wake up everybody; processes all try P again.
  - Is this “busy-waiting”?
- Step 2:
  - Label each process with the semaphore it is waiting for.
  - Then, just wake up relevant processes.
- Step 3:
  - Just wake up a single process.
- Step 4:
  - Add a queue of waiting processes to the semaphore, wake up FCFS.
  - On failed P, add to queue.
  - On V, remove from queue.

```
typedef struct {

   int count;

   queue q;

} Semaphore;
```

```
P( Semaphore *sem ) {
   interruptsDisable();
   if ( sem->count > 0 ) {
      sem->count--;
      interruptsEnable();
   }
   else {
      Move process fm ready list to sem->q;
      interruptsEnable();
      dispatcher();
   }
} /* P */
```

```
V( Semaphore ) {
   interruptsDisable();
   if ( sem->q not empty ) {
      Move first process fm sem->q to
         ready list
      dispatcher();
   }
   else
      sem->count++;
   interruptsEnable();
} /* V */
```

- There are several trade-off's implicit here:
  - How many processes are in the system?
  - How much queueing on semaphores, storage requirements, etc.?
  - Most important thing: <u>avoid busy-waiting</u>.

- What do we do in a multiprocessor to implement P's and V's? Cannot just turn off interrupts to get low-level mutual exclusion.
  - Turn off all other processors?
  - Use atomic read and write, as in too-much-milk?
- In a multiprocessor, there will have to be busy-waiting at some level: cannot block without mutual exclusion.
  - Marking the process as blocked <u>and</u> blocking it must be atomic.
- Most machines provide some sort of atomic read-modify-write instruction. Read existing value, store back in one atomic operation.
  - Example: Test-and-set — TAS
    - Implemented initially by IBM, later by many others.
  - The TAS instruction sets the value to one, but returns <u>old</u> value.
    - Use ordinary write to set back to zero.

```
la   $s2, memAddr
lw  $v0, 0($s2)
addi  $s1, $zero, 1
sw  $s1, 0($s2)
```

  - Using test-and-set for mutual exclusion:
    - Like a binary semaphore in reverse (but does not include waiting).
    - 1 = someone else is already using it.
    - 0 = okay to proceed.
    - By definition, two (or more) processes are prevented by TAS from getting a 0-to-1 transition simultaneously.

- Test-and-set is tricky to use:
  - Cannot get at it from HLLs.
  - Typically, must use a routine written in assembler.

- Read-modify-write's may be implemented:
  - Directly in memory hardware (e.g., IBM 360), or
  - In the processor by refusing to release the memory bus (PDP-11).

- Using test-and-set to implement semaphores in a multiprocessor:
  - For each semaphore, keep a test-and-set integer in addition to the semaphore integer and the queue of waiting processes.
  - This integer is often called a *spin-lock* since the processes will busy-wait (or *spin*) on it.

```
typedef struct {
   int count;
   queue q;
   int test;      // set to 0 by SemCreate
} Semaphore;
```

```
P( Semaphore *sem ) {
   while (1) {
      interruptsDisable();
      while ( TAS( &sem->test ) != 0 )
         ; /* do nothing */
      if ( sem->count > 0 ) {
          sem->count--;
          break;
      }
      Enqueue process on sem->q
      sem->test = 0; /* ordinary assignment */
      interruptsEnable();
      dispatcher(); /* cannot return! */
   }
   sem->test = 0;
   interruptsEnable(); /* return okay */
} /* P */
```

```
V( Semaphore *sem ) {
   interruptsDisable();
   while ( TAS( &sem->test ) != 0 )
      ; /* do nothing */
   sem->count++;
   if ( sem->q is not empty ) {
      Dequeue first process from sem->q;
      Wake it up;
   }
   sem->test = 0; /* ordinary assignment */
   interruptsEnable();
} /* V */
```

- Important point: Implement some mechanism once, very carefully. Then, always write programs that use that mechanism.
  - Importance of *layering*.
- Why do we still have to disable interrupts in addition to using test-and-set?
  - Semaphores synchronize *processes*.
  - An interrupt handler is not a separate process; it is a procedure run by the interrupted process.
  - What if the interrupted process was in a critical section and the interrupt handler tries to enter the same section? Neither will make progress: deadlock!
- Semaphores cannot be used to provide mutual exclusion between interrupt handlers and processes.
  - On a uniprocessor, you must disable interrupts.
  - On a multiprocessor, you must disable interrupts and busy-wait (spin-lock).
  - Different forms of mutual exclusion complicate the code.
- Interrupt handlers can only V semaphores (unless you code very carefully).
  - Provides a limited form of scheduling