Race Conditions, Critical Sections and Semaphores

- In a multiprogrammed system, there are several processes "active" at once.
  - Even a single job can create multiple processes (as in the Lab project using FORK).
    - Only one process can be executing at any instant in time given a uni-processor.
    - Generally, programmers have no control over when processes are swapped.
      - Each process could potentially get interrupted after any instruction.
      - Swapping occurs under the control of the CPU Scheduler; not programmer.
      - The OS Scheduler determines when each process runs; not the programmer.
    - Processes interact, not just in sharing of resources as in the deadlock problem.
    - They also need to communicate and coordinate with each other.
  - Here, we examine problems that can occur when Interprocess Communication (IPC) happens through the use of Shared Variables.

- Race Conditions
  - A potential IPC problem when two or more processes interact via shared data.
  - Final outcome depends on the exact instruction sequence of the processes.
    - Instruction-by-instruction sequence determined by the OS CPU Scheduler.
    - Depends on which process "wins" the "race".
  - Extremely difficult to debug because of its transient nature; Lurking bugs possible.

- "The *too much milk* problem"

time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy Milk
3:50		Arrive home, put up milk
3:50		Oh no!

- Example of Race: The ATM "Withdraw at the same time from a joint account" Problem

Balance = $300

ATM -1	**ATM-2**
$300 ←Check Balance	
	Check Balance → $300
??? ←Withdraw $300	**Withdraw $300 → ???**

- Example of Race: Print Spooler

When process wants to print a file, it enters the filename onto the spooler list.

Process P is printer. It checks spooler list periodically and prints the jobs.

Spooler list has consecutively numbered slots; each can hold a filename.

There are global shared variables associated with the spooler list.

LIST: Large linear list where LIST(i) holds a filename in the i^{th} slot.

IN: Points to the next free slot where a filename can be put in spooler list.

OUT: Points to the next file (slot) in the spooler list to be printed out by P.

Assume that two processes, A and B, want to print at the "same" time.

<u>Each</u> process must perform the following sequence of instructions:

```
next_free_slot = IN                          /* read value of IN
LIST (next_free_slot) = <filename>           /* write filename in the free slot
IN = next_free_slot + 1                      /* update value of IN
```

Global Shared variables are shown in capital letters.

Local variables (one set exists for each process) are shown in lower case.

Assume LIST (4, 5 and 6) already contain filenames queued for printing.

So IN = 7; OUT = 4 and execution sequence is as follows (time flows down):

A: next_free_slot$_A$ = 7		/* A reads IN of 7
	B: next_free_slot$_B$ = 7	/* B reads IN of 7
	B: LIST(7) = <filename$_B$>	/* B writes filename
	B: IN = 8	/* B updates IN
A: LIST(7) = <filename$_A$>		/* A writes filename
A: IN = 8		/* A updates IN

Problem: Process A writes over Process B's filename in LIST(7) because Process B began using a shared variable before Process A was done with it.

- **Critical Sections.**

  First proposed by Dijkstra in 1965.

  A Critical Section of a program is where global shared memory is being accessed.

  Being inside a critical section is a special status accorded to a process.

  Process has exclusive access to shared modifiable data while in critical region.

  All other processes needing access to that shared data are kept waiting.

  Therefore, critical sections must:

  Execute as quickly as possible.

  Be coded carefully to reduce any possibility of errors (e.g., infinite loops).

  Ensure termination housekeeping if process in critical region aborts.

  Must release mutual exclusion so other processes can enter region.

The majority of the time, a process performs only non-critical work on local vars.

Therefore critical sections of a program can be made very small.

Goal is to enable interacting asynchronous concurrent processes to yield correct results independent of their execution speeds and when they are swapped.

To avoid race conditions, mutual exclusion must be enforced within critical sections.

Prohibits more than one process from accessing shared memory at same time.

If no two processes enter their critical sections at same time, no race conditions.

For correct and efficient operation using shared data, a solution must ensure that:

1) No two processes can be inside their critical sections at same time.

2) No assumptions needed about relative process speeds or number of CPUs.

3) No process stopped outside its critical section can block other processes.

4) No process can be indefinitely postponed from entering its critical section.

- **Mutual Exclusion.**

Designing Primitive Operations for achieving mutual exclusion requires help from OS

Only needed when processes access shared modifiable data (in critical region).

Concurrent execution OK if two processes do not conflict with one another.

That is, Race Condition is not a problem if they do not share data.

Just like Deadlock is not a problem if processes do not share any resources.

We will look at Semaphores to ensure that only one process is in a critical section

- **Semaphores (Dijkstra, 1965)**

  The analogy:

  A semaphore can act like a gate into a restricted area.

  Status of the gate is either open (raised) or closed (lowered).

  When a process needs access, it executes a P operation.

  If gate is open, process enters and closes gate behind it.

  Otherwise, if gate is closed, process must wait until gate is reopened.

  Once process has entered, used shared var, and exits critical section, it calls V

  V opens the gate and allows (signals) one waiting process to enter.

  If no processes waiting, gate is left open for next process needing access.

  History: Dutch words for 'wait' and 'signal' have initial letters P and V respectively.

  Software Implementation:

  Let S be a semaphore.

  A semaphore is simply an integer value.

  Its value can only be altered via operations P(S) and V(S).

  An important aspect of semaphores is that P and V are executed atomically.

  That is, P and V are made indivisible, generally with OS and H/W support.

  Executed by the OS kernel with all interrupts disabled.

  Indivisible: Once started, they will be completed without interruption.

  This prevents two processes from executing P or V operations at the same time.

  No other process can access the semaphore until the operation is completed.

Also, inherent and built-into P and V are OS-level wait and signal capabilities.

Processes are put to sleep (swapped out of CPU) while waiting for a signal.

This eliminates any CPU inefficient busy waiting (e.g. constantly checking lock).

[Note: Algorithm for P(S) and V(S) is only one of many alternatives in literature]

P(S) acquires permission to enter a critical region.

Alternate names for P include Wait and Down.

When a process executes P(S) the following steps are done atomically:

S is *decremented* by 1, and

IF S .*LT*. 0, then the calling process is stopped and put on a waiting queue.

It remains blocked until a V(S) operation by another process releases it.

ELSE calling process continues

V(S) records the termination of a critical region.

V(S) must reactivate one of the waiting processes (if any).

Alternate names for V include Signal and Up.

When a process executes V(S) the following steps are done atomically:

S is *incremented* by 1, and

IF S .*LE*. 0, then one process from the waiting queue is removed from the queue

The waiting process from the queue is allowed to resume execution.

The calling process which invoked V(S) can also continue execution.

- **Semaphores can be used in several different ways.**

  Different applications require different initial values for, and numbers of, Semaphores

  A semaphore which has a maximum value of one is called a binary semaphore.

  Binary Semaphore often referred to as MUTEX (MUTual EXclusion).

  Two processes can implement mutual exclusion by using a binary semaphore.

  Critical sections are bracketed by P(S) and V(S).

  P(S) is the entry or opening bracket; V(S) is the exit or closing bracket.

  For two processes with a binary semaphore:

  If S = 1, then neither process is executing its critical section.

  If S = 0, then one process is executing its critical section.

  If S = -1, then one process is in its critical section and other is waiting to enter.

  - Using Semaphores to Wait for something to be done.

  Process A will wait until Process B reaches a certain point.

  SEM is initialized to 0

Process A	**Process B**
.	.
Start (Process B)	**<step that A is waiting for>**
P (SEM)	**V (SEM)**
.	.
.	.

- Using Semaphores for Rendezvous

No matter which process reaches the rendezvous point first,

it will wait for the other process to catch up before proceeding.

This technique allows the two processes to re-synchronize.

Two Semaphores are required: SEM1 and SEM2, both initialized to 0

Process A	Process B
.	.
V (SEM1)	V (SEM2)
P (SEM2)	P (SEM1)
.	.
.	.

If Process A arrives first: (Would be Symmetrical if B arrives first)

Time	*Process*	*Action*	*Waiting*	*SEM1*	*SEM2*
				0	0
1	A	V(SEM1)			
				1	0
2	A	P(SEM2)			
			A	1	-1
3	B	V(SEM2)			
				1	0
4	B	P(SEM1)			
				0	0

- Semaphores for Mutual Exclusion: SEM initialized to 1

Process A

```
Loop_A:
    <non-critical>
P (SEM)
    <critical section>
V (SEM)
    <non-critical>
GOTO Loop_A
```

Process B

```
Loop_B:
    <non-critical>
P (SEM)
    <critical section>
V (SEM)
    <non-critical>
GOTO Loop_B
```

Time	*Process*	*Action*	*In C.S.*	*Waiting*	*SEM*
					1
1	A	P(SEM)			
			A		0
2	B	P(SEM)			
				B	-1
3	A	V(SEM)			
			B		0
4	B	V(SEM)			
					1

- Example: Four Process Mutual Exclusion with SEM initialized to 1.

Process A	Process B	Process C	Process D
P (SEM)	P (SEM)	P (SEM)	P (SEM)
<critical>	*<critical>*	*<critical>*	*<critical>*
V (SEM)	V (SEM)	V (SEM)	V (SEM)

Time	*Process*	*Action*	*In C.S.*	*Waiting*	*SEM*
					1
1	A	P(SEM)			
			A		0
2	A	V(SEM)			
					1
3	B	P(SEM)			
			B		0
4	C	P(SEM)			
				C	-1
5	D	P(SEM)			
				C, D	-2
6	B	V(SEM)			
			C	D	-1
7	C	V(SEM)			
			D		0
8	D	V(SEM)			
					1

- **Using Semaphores for a Producer-Consumer (Bounded Buffer) System**

Two processes must communicate and coordinate using a common buffer.

Producer generates and places items in a shared buffer.

Consumer takes items off of the shared buffer and uses (consumes) them.

This problem is not just one of mutual exclusion, it is also involves synchronization.

The producer and consumer proceed asynchronously and at different speeds.

If Producer runs faster than the Consumer, the buffer may fill up.

Producer must wait if buffer is full until the Consumer empties one slot.

If Consumer runs faster than the Producer, the buffer may be empty.

Consumer must wait if buffer is empty until the Producer fills one slot.

Three semaphores are needed:

1) EMPTY will count the number of buffer slots not yet filled (i.e., empty)

EMPTY is initialized to N, the total number of slots in buffer, 2 in ex. below.

Producer must wait at P(EMPTY) if there are no more empty slots.

Consumer increments EMPTY via V(EMPTY) after consuming a slot.

2) FULL will count the number of buffer slots that have been filled.

FULL is initialized to 0 since entire buffer is initially empty.

Consumer must wait at P(FULL) if there are no more filled buffer slots.

Producer increments FULL via V(FULL) after producing (filling) a slot.

3) MUTEX guarantees that only one process is operating on buffer pointers.

MUTEX is initialized to 1 and guards <*Fill Buffer*> and <*Consume Buffer*>.

Producer:

```
<non-critical>
P (EMPTY)
P (MUTEX)
    <Fill Buffer>
V (MUTEX)
V (FULL)
    <non-critical>
GOTO Producer
```

Consumer:

```
<non-critical>
P (FULL)
P (MUTEX)
    <Consume Buffer>
V (MUTEX)
V (EMPTY)
    <non-critical>
GOTO Consumer
```

Time	*Producer*	*Consumer*	*Empty*	*Full*
			2	0
1		P(FULL) Blocked (Buffer empty)		
			2	-1
2	P(EMPTY)			
			1	-1
3	V(FULL) Unblocks Consumer			
			1	0
4	P(EMPTY)			
			0	0
5		V(EMPTY)		
			1	0
6	V(FULL)			
			1	1
7	P(EMPTY)			
			0	1
8	V(FULL)			
			0	2
9	P(EMPTY) Blocked (Buffer full)			
			-1	2
10		P(FULL)		
			-1	1
11		V(EMPTY) Unblocks Producer		
			0	1

- Using Semaphores for The Readers and Writers Problem

Another "Classic" IPC Problem which models access to a shared Data Base.

Two types of processes, readers and writers, compete for access to the Data Base.

Access Protocol:

Multiple readers can read the Data Base concurrently.

However, Only a single writer can write to the Data Base at any one time.

Also, if a writer is writing to the Data Base, no readers can be allowed access to it.

Implementation:

The first and last reader to enter or leave the critical region execute special duties.

RC counts the number of readers to allow us to identify the first and last reader

Proposed Solution below uses two semaphores:

1) DB is initialized to 1 and guarantees that access for writing is exclusive.

Used by the writer before and after writing to protect critical write section.

Also used by the first reader to determine if a writer is currently writing.

If so, it and all other readers behind it wait; otherwise reader enters.

Last reader to exit from critical section must release writer from waiting.

2) MUTEX, initialized to 1, guarantees that the shared variable RC is protected.

Ensures only one reader at a time can update the reader_counter variable, RC.

Reader:

```
P (MUTEX)                    /* Only one reader can update RC at a time
    RC = RC + 1              /* Increment number of readers upon reader entry
    IF (RC = 1) THEN P(DB)   /* 1st reader ensures mutual exclusion with writers
V (MUTEX)                    /* End critical section on RC update

    <read_data_base>         /* Critical only to writers; OK to have many readers

P (MUTEX)                    /* Only one reader can update RC at a time
    RC = RC - 1              /* Decrement number of readers upon reader exit
    IF (RC = 0) THEN V(DB)   /* Last reader must release any waiting writer
V (MUTEX)                    /* End critical section on RC update

GOTO Reader
```

Writer:

```
P (DB)                       /* Only one writer at a time and only if no readers
    <write_data_base>        /* Critical Section
V (DB)                       /* End critical section on write_data_base.

GOTO Writer
```

Note: The Reader-Writer Solution above implicitly gives readers priority over writers.

Only the first reader has to compete with any writers to gain access.

Other readers will pass directly into the Critical Region (bypassing the already waiting writer) as long as there is at least one other reader in CR

No writers are allowed to begin writing if there are any readers reading.

So, writers can be indefinitely postponed given an infinite supply of readers.

- **Use Care with Semaphores and Locks in General**
  - **They have Overhead**
  - **They Enforce Sequential Access to a Shared Resource by even Parallel Processes**
  - **They can cause Lack of Fairness**
    - **Not all processes are given "equal, fair" access to a "shared" resource**
    - **Extreme case can lead to Starvation**
      - **Some process gets deferred forever (as writers could in above solution)**
  - **They can lead to Deadlock:**

Process A	**Process B**
P (Sem1)	**P (Sem2)**
P (Sem2)	**P (Sem1)**
V (Sem2)	**V (Sem1)**
V (Sem1)	**V (Sem2)**