Process Synchronization

Consider two threads using and modifying a shared global variable. What problems could occur?

Example: a bank account balance (a shared global variable)

Balance $200
A: Deposit $10
B: Deposit $10,000
Process Synchronization

A: get Balance, $200
B: get Balance, $200
B: 200 + 10,000
B: write balance, $10,200
A: 200 + 10
A: write balance, $210

Not an atomic action
- Can lead to race condition
  which can lead to invalid data.
Process Synchronization

The balance is a shared resource and we must provide a way to ensure mutual exclusion of it.

That is give one and only one thread access to balance at a time. Lets call the code that updates/uses a shared resource a critical section. A code segment that must be only executed by one thread at a time.
Process Synchronization

Want to ensure mutual exclusion of the critical section.
- only let one thread execute their critical section at a time.

Can avoid race conditions if no two processes are in their critical section at the same time.
4 conditions to avoid race cond.

1. No 2 processes may be simultaneously inside their critical section.

2. No assumptions may be made about speeds or the number of CPUs.

3. No process running outside its critical section may block other processes.

4. No process should have to wait forever to enter its critical section.
Inter Process Communication

What does 'ls -l | less' do?
How does it work?

Stdout of ls is piped to stdin of less by the shell.

A | B | C | D

IPC = Inter Process Communication.
Inter Process Communication

One process communicates with another.

Issues:
- How to pass info to another process (easy for threads).
- make sure processes do not get in each others way. (also a problem for threads)
- proper sequence.

Look at some ways to provide MutEx
Disable Interrupts

Simplest solution
Processes disable interrupts at beginning of critical section and re-enable them when leaving CS.

Disable Interrupts
do the critical section
Enable interrupts

With interrupts disabled switching to another process is not possible.
Disable Interrupts

Not a wise idea to give user processes the ability to disable interrupts.

Also does not work on multiprocessor as interrupts would be disabled just for one cpu.

Sometimes useful for the kernel to do this.
Lock Variables

Take on two values, true/false, 0/1, locked/unlocked.

If (!locked)

Critical Section

{ lock = 1;

Critical Section

lock = 0;

} else

{ wait;

then try again;

}
Lock Variables

Seems like a simple solution.

But testing and setting the lock variable is a race condition itself!!
Strict Alternation

Spin lock with busy waiting.
Busy waiting wastes cpu time - use only when wait expected to be short

while ( 1 )
{
    while(turn != 0 ) ;     while(turn != 1) ;
    C.S.
    turn = 1;  
}

Processes must alternate - not a good solution
Peterson's Solution

Call enter_region() with process # 0 or 1. Will wait if need to, then do C.S and call leave_region().

#define FALSE 0
#define TRUE 1
#define N 2
int turn;        /* whose turn is it? */
int interested[N]; /* all values initially FALSE */

void leave_region(int process)
{
    interested[process] = FALSE;
}
Peterson's Solution

void enter_region(int process)
{
    int other;    /* number of the other process */

    other = 1 - process;
    interested[process] = TRUE; /* now interested*/
    turn = process;                /* set flag */
    while (turn == process && interested[other] == TRUE )
        ; /* null statement */
}
Test and Set Lock Instruction

Test and Set Lock hardware instruction
- can test and set a lock variable in one atomic instruction.

**enter_region:**
TSL Register, Lock ; copy lock to reg and set lock
CMP Register, #0 ; see if 0
JNE enter_region ; non-zero, lock was set, loop
ret

**leave_region:**
move Lock, #0 ; store a 0 in lock
ret
Peterson's solution and TSL need busy waiting which wastes cpu and can have undesirable effects.

Consider a high priority process H, which is scheduled to run whenever it is ready and a low priority process L.

What happens when L is in critical section and H starts a busy loop?
Sleep and Wakeup

What happens when L is in critical section and H starts a busy loop?

H runs, but L never gets to leave C.S. Solution is to let L run instead of H - priority inversion problem

Sleep will block until another process wakes it up.
Producer-Consumer Problem

Two processes (threads) share a fixed size buffer.
The producer puts things in and the consumer takes things out.
If buffer is full producer must wait for a slot to open up.
If buffer is empty consumer must wait until a slot is filled.
Need a variable, count, to keep track of number of slots filled in buffer.
Size of buffer is N (bounded buffer)
void producer()
{
    int item;
    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1)      // was buffer empty
            wakeup(consumer);
    }
}
void consumer()
{
    int item;
    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) // was buffer full?
            wakeup(producer);
        consume_item(item);
    }
}

Producer-Consumer Problem

Problem with this code:

Fatal race condition on count variable!

If buffer is empty consumer reads 0, producer then runs and increments count, calls wakeup. But consumer is not sleeping yet and wakeup is lost.
Semaphores

Come from Dijkstra in 1965.

Two or more processes can cooperate by sending signals.

A semaphore is a special variable with two primitive operations:

signal – v (verhogen) – to increment (up)
wait – p (proberan) – to test (down)
Semaphores

$s$ is a semaphore (nonnegative integer)

\begin{align*}
up(s) &= \text{signal}(s) : [ s = s + 1 ] \\
down(s) &= \text{wait}(s) : [ \text{while}(s==0) \{ \text{wait} \}; \\
& \quad s = s - 1; ]
\end{align*}

These are atomic operations
- cannot be interrupted
- each treated as an indivisible step
Semaphores

General, counting semaphore
- 0, 1, 2, 3, ...
A binary semaphore takes on values of 0 or 1 only.

wait/down test and blocks if not positive
signal/up gives the go ahead signal
Blocked processes wait in a queue
- a FIFO is the fairest policy for removal
(a strong semaphore)
A weak semaphore doesn't specify order
Semaphores

Ex: Bank account problem

```c
semaphore mutex = 1;
deposit (amount)
{
    /* enter C.S. */
    wait(mutex);
    balance += amount;
    /* exit C.S. */
    signal(mutex);
}
```

Semaphores are normally implemented with system calls with interrupts disabled.
Semaphores

Ex of 3 processes accessing shared data protected by a semaphore (lock).

A
  wait(lock)
  signal(lock)

B
  wait(lock)
  signal(lock)

C
  wait(lock)
  signal(lock)
Semaphore soln to producer-consumer problem

Three semaphores:
- full – count of slots filled, init 0
- empty – count of empty slots
- mutex – for mutual exclusion of buffer
  init slots in buffer
  init 1
  binary semaphore

full and empty are used for synchronization.
Semaphore soln to producer-consumer problem

#define N 100

semaphore mutex = 1;
semaphore empty = N; /* count empty buffer slots */
semaphore full = 0; /* count full buffer slots */
void producer()
{
    int item;
    while (TRUE) {
        item = produce_item();
        wait(empty);  // decrement empty
        wait(mutex);
        insert_item(item);  // critical section
        signal(mutex);
        signal(full);  // increment full
    }
}
Semaphore soln to producer-consumer problem

```plaintext
void consumer()
{
    int item;
    while (TRUE) {
        wait(full); // decrement full
        wait(mutex);
        item = remove_item(); // critical section
        signal(mutex);
        signal(empty); // increment empty
        consume_item(item); // use it
    }
}
```
Mutexes

A semaphore not used for counting. Has locked and unlocked states.

mutex_lock() and mutex_unlock()

mutex_trylock() - either locks or returns a failure instead of blocking.
Monitors

Semaphores are difficult to use in programs correctly. A **monitor** is a higher-level synchronization primitive that is easier to use. A group of functions, variables, and data structures make up a monitor. - synchronized methods in java

A process/thread can call a function in a monitor, but not access the internal data directly.
Monitors

Only one process can be active in a monitor at a time -> **mutual exclusion**

Compiler must take care in when monitor functions are called.
At beginning of monitor check to see if any processes are using it.
If there are suspend, else allow to run.
Mutual exclusion is easy, but what about blocking when a process needs to wait?
Wait and signal operations on conditions variables can be used.
Message Passing

Two primitives (system calls) 
send(dest, mesg) and 
receive(src, mesg) 
- could block when no mesg avail or 
return an error code.

Can be used over a network 
- messages could be lost 
  - need to ack, have retransmits 
- use sequence numbers in case ack 
  is lost and retransmit happens.
Message Passing

Authentication is an issue

Need a way to address messages
- Add message to process
- have a mailbox – a buffer to store messages.

Barriers – Used for groups of processes (more than two)
A process cannot continue until all processes have reached the barrier