```
Sources of concurrency:
   − multiple processes
   − one process with multiple threads
   − multiple physical processors
   − network

Synchronization:

Given 2 more more processes (or threads) which share a resource (e.g.,
variable or device), we must often synchronize their activity.

Must satisify to one degree or another the concepts of mutual exclusion,
progress, and bounded waiting.

Example: consider only 2 processes

critical section (<cs>): instructions which access shared resource

We must establish mutual exclusion: no 2 processes can be
in their <cs> at the same time.

process producer {

    while (true) {

      while (count == BUFFER_SIZE);
      ++count;
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
   }
}

process consumer {

   while (true) {
      while (count == 0);
      −−count;
      item = buffer[out];
      out = (out − 1) % BUFFER_SIZE;
   }
}

Assume count = 5 and both producer and consumer execute the statements ++count
and −−count.

Results? count could be set to 4, 5, or 6 (but only 5 is correct).

reg_1 = count
reg_1 = reg_1 + 1
count = reg_1

reg_2 = count
reg_2 = reg_2 − 1
count = reg_2

principle of ‘atomicity’

t_0: producer executes reg_1 = count     [reg_1 = 5]
t_1: producer executes reg_1 = reg_1 + 1 [reg_1 = 6]
t_2: consumer executes reg_2 = count     [reg_2 = 5]
t_3: consumer executes reg_2 = reg_2 − 1 [reg_2 = 4]
t_4: producer executes count = reg_1     [count = 6]
t_5: consumer executes count = reg_2     [count = 4]

race condition: a situation where multiple processes access and
manipulate the same data concurrently and the outcome of the exececution
depends on the order in which the instructions execute.

Operating system kernel code itself can have race conditions.
```

```

preemptive vs. nonpreemptive kernels

This is called ‘the critical section problem’

Possible solutions?
   − disable interrupts during execution of <cs>

      process p_i {
         while (true) {
            // disable interrupts (a system call)
            // critical section
            // enable interrupts (a system call)
            // remainder section
         }
      }

      − degrades efficiency
      − not possible multiprocessor systems
   − synchronization

A solution must satify three requirements:

   − mutual exclusion: only one process may execute its critical section at
                       once; prevent concurrent access to shared objects to
                       preserve the consistency of the object.

   − progress: if no process is executing in its critical section and some
     processes wish to enter their critical sections, then only those
     processes not executing in their remainder sections can participate
     in the decision on which process will enter its critical section next,
     and this decision cannot be postponed indefinitely.

   − bounded waiting: this is a limit on the number of times other processes
     are allowed to enter theur critical section after a process has made
     a request to enter its critical section and before that request is granted.

Basic idea in synhronization: need locks in one form or another.

while (true)
   // entry section; acquire lock
   // critical section
   // exit section; release lock
   // remainder section
}

Peterson’s Solution

shared data:

int turn;
boolean flag[2];

process p_i {

   while (true) {
      flag[i] = true; // I am in my <cs>
      turn = j;       // but I give p_j priority

      // as long as p_j wants access and it is p_j’s turn, I do no−op
      while (flag[j] && turn == j);

      // critical section

      // I am no longer in my <cs>
      flag[i] = false;

      // remainder section;
   }
```

```
}

process p_j {

   while (true) {
      flag[j] = true; // I am in my <cs>
      turn = i;       // but I give p_i priority

      // as long as p_i wants access and it is p_i’s turn, I do no−op
      while (flag[i] && turn == i);

      // critical section

      // I am no longer in my <cs>
      flag[j] = false;

      // remainder section;
   }
}

Peterson’s solution guarantees mutual exclusion, progress, and bounded
waiting.

What is the problem with Peterson’s solution?

busy waiting: the waiting process wastes CPU cycles; in a uniprocessor
system, the process waits until its quantum expires

Hardware support: two new instructions, both versions of a read−modify−write
instruction

   − test and set

   boolean test−and−set (boolean* target) {
      boolean temp = *target;
      *target = true;
      return temp;
   }

   boolean occupied = false;

   while (true) {

      while (test−and−set (&occupied));

      // critical section

      occupied = false;

     // remainder section
   }

   problems? starvation

   − swap

   boolean occupied = false;

   void swap (boolean* x, boolean *y) {
      boolean temp = *x;
      *x = *y;
      *y = temp;
   }

   boolean p_i_must_wait = true;

   while (true) {

      do
```

```
         swap (&p_i_must_wait, &occupied);
      while (p_i_must_wait);

      // critical section

      p_i_must_wait = true;
      occupied = false;

     // remainder section
   }

Semaphores

Classical Problems of Synchronization

Monitors

Types of solutions:
   − hardware
      − disable interrupts (will not work in a multiprocessor system)
      − test and set instruction
      − swap instruction
   − software
      − semaphores
      − monitors
      − conditional critical sections

Semaphores vs. Monitors

Semaphores
   − wait does not necessarily block
   − signal unblocks (without delay) or increments (remembered)

Monitors
   − mutual exclusion with the monitor boundaries

   − wait always blocks

   − signals unblocks (but then unblocked process must acquire monitor lock), or
     has no effect

   − limitations
      − only 1 process at a time in the monitor −> absence of concurrency
      − critical section is seperate from the monitor −> weakens encapsulation

      − possibility of deadlock with nested monitor calls
```