SWEN-220
Mathematical Models of Software

Process Synchronization
Critical Section & Semaphores
Topics

• The critical section
• Synchronization using busy-wait
• Semaphores
The Critical Section

- Processes can have a critical and non-critical section

- **Non-critical** section works with **local resources**

- **Critical** section works with **global resources**

- In general, processes may halt in their non-critical section, but not in their critical section
The Critical Section Problem

• Need to ensure Safety & Liveness

• Mutual exclusion
  – At most one process is executing its critical section at any time

• No Deadlock
  – It is impossible to reach a state in which some processes are trying to enter their critical sections, but no process is successful

• Fairness
  – If any process is trying to execute its critical section, then eventually that process is successful
The Critical Section Example

bool wantP = false, wantQ = false;

active proctype P(){
do
:: printf("noncritical sec P\n");
  wantP = true;
  printf("critical section P\n");
  wantP = false
  od;
}

active proctype Q(){
do
:: printf("noncritical sec Q\n");
  wantQ = true;
  printf("critical section Q\n");
  wantQ = false
  od;
}
Solving the Critical Section Problem

• One of the main issues is that \textit{wantP} and \textit{wantQ} is never read by the other process

• What can we do?

• One way to try and solve the problem is have process P read \textit{wantQ} and process Q read \textit{wantP}
Synchronization using Busy-wait

• Make the process wait until the other process is finished executing its critical section

```c
do
  :: printf("noncritical sec P\n");
  do
    :: !wantQ -> break;
    :: else -> skip;
  od;
  wantP = true;
  printf("critical section P\n");
  wantP = false
od;
```
Problem with using Busy-wait

• The loop performs no useful computation. It is repeatedly evaluating the expression until the other guard becomes true

• This wastes cycles and is unacceptable, except for multiprocessors, which have a large number of cycles
Alternative to using Busy-wait, Blocking

• Most computer systems are based on “blocking”

• Synchronization by blocking is used in operating systems that implement multitasking.

• Synchronization is achieved by having a process execute an operation that causes it to block, thus enabling another process to run
do
:: printf("noncritical sec P\n");
do
:: !wantQ -> break;
:: else -> skip;
od;
wantP = true;
printf("critical section P\n");
wantP = false
od;

But we can simply this even further!
Executability of Statements in Promela

• An expression statement is executable in Promela, if and only if it evaluates to true

```plaintext
do:: printf("noncritical sec P\n");
  do
    :: !wantQ -> break;
  od;
wantP = true;
printf("critical section P\n");
wantP = false
od;
```

• Assignment statements and printf statements are always executable
Synchronization using Blocking

• A process is blocked until the other process is finished executing its critical section

do
:: printf("noncritical sec P\n");
    atomic{
        !wantQ;
        wantP = true;
    }
    printf("critical section P\n");
    wantP = false
od;
Issue with using Blocking

• Blocking in these previous examples requires obtaining a “lock” (boolean flag) before entering a critical section of the process.

• The lock is often referred to as a “mutual exclusion lock”, or **mutex**.
  – Example: Think of a public restroom that requires obtaining a key.

• What happens if we want to synchronize more than two processes, or allow more than one to be in the critical section?
  – Example: Think of multiple vehicles crossing a bridge that has a limit of n vehicles being on the bridge at the same time.
1. A semaphore can be initialized to any integer value, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one).

2. You cannot read the current value of the semaphore.

3. When a process decrements - \texttt{down()} or \texttt{wait()}, the semaphore, if the result is negative, the process blocks itself and cannot continue until another process increments the semaphore.

4. When a process increments the semaphore – \texttt{up()} or \texttt{signal()}, if there are other processes waiting, one of the waiting processes gets unblocked.
Semaphore - Usage

• In SPIN, a semaphore is a variable of type byte, that has the two *atomic* operations
  – **decrement, down, or wait**(sem): operation is executable when sem>0; executing the operation decrements sem
  – **increment, up, or signal**(sem): operation is always executable; executing the operation increments the value of sem

• Historical note: Semaphores were proposed by Edgar Dijkstra in the 1960’s, being Dutch he used:
  – **V()** Verhogen – “to increment” for up() or signal()
  – **P()** Proberen – “to test” for down() or wait()
Semaphore Example

do
:: printf("noncritical sec in P = %d\n", _pid);
  atomic{
    sem > 0;
    sem--;
  }
  printf("critical section in P = %d\n", _pid);
  sem++;
od;
Semaphore “functions”

```c
#define semaphore byte // Pseudo-type semaphore

semaphore sem = 1; // Declare the semaphore (sem), initialize to max
// processes sharing protected resource

/* Up and down “functions” on semaphores.
 * down blocks if the semaphore is 0 on entry.
 */
inline unlock(s) { s++ }
inline lock(s) { atomic { s > 0; s-- } }

do:: printf("noncritical sec in P = %d\n", _pid); // _pid is process identifier (0-n)
    lock(sem); // _pid
    printf("critical section in P = %d\n", _pid);
    unlock(sem);
od;
```
Signaling Semaphores

• Semaphores can also be used as a “signaling" mechanism between processes. Java threads have this behavior via `wait()` & `notify()`.

• A process will reach a point in its execution where it needs to block and await a signal from another process to proceed.

• The waiting process will perform a `down(sem)` and then proceed when a notifying process performs an `up(sem)`.

• Note that signaling semaphores are initialized to zero.
The same semaphore “functions” used for mutex semaphores are also used for signaling semaphores with the exception being that signalling semaphores are initialized to zero.

```
semP = 0;
semQ = 0;

proctype P(){
    do  // do some stuff, then block waiting for a signal from Q()
         wait(semP);  // Note sem starts at zero, so this blocks P()
    // continue here when signaled by Q()
         notify(semQ);   // Unblock Q()
    od;
}

proctype Q(){
    do  // do some stuff, then signal P() to proceed
         notify(semP);  // This unblocks P()
    // continue here after signaling P()
         wait(semQ);   // Block here until notified by P()
    od;
}
```