Process Synchronization Mechanisms

Tom Kelliher, CS 311

Mar 19, 2012

Announcements:

From last time:

1. CPU scheduling.

Outline:

1. Critical sections and cooperating processes.
2. Cooperating processes (review).
3. A hardware solution to the C. S. problem.
4. Semaphores.

Assignment: Read Chapter 6.

1 Critical Sections and Cooperating Processes

What is a critical section?

> *The overlapping portion of cooperating processes, where shared variables are being accessed.*

Not all processes share variables: independent processes.

Cooperating/independent processes.

Necessary conditions for a solution to the c.s. problem:

1. Mutual Exclusion — if P_i is executing in one of its critical sections, no P_j, $j \neq i$, is executing in its critical sections.
2. Progress — a process operating outside of its critical section cannot prevent other processes from entering theirs; processes attempting to enter their critical sections simultaneously must decide which process enters eventually.
3. Bounded Waiting — a process attempting to enter its critical region will be able to do so eventually.

Assumptions:

1. No assumptions made about relative speed of processes
2. No process may remain in its critical section indefinitely (may not terminate in its critical section)
3. A memory operation (read or write) is atomic — cannot be interrupted. For now, we do not assume indivisible RMW cycles.

Classic example: the *producer/consumer* problem (aka bounded buffer):

Global data:

```
const int N = 10;

int buffer[N];
int in = 0;
int out = 0;
int full = 0;
int empty = N;
```

Producer:

```
while (1)
{
   while (empty == 0)
      ;

   buffer[in] = inData;
   in = ++in % N;
   --empty;
   ++full;
}
```

Consumer:

```
while (1)
{
   while (full == 0)
      ;

   outData = buffer[out];
   out = ++out % N;
   --full;
   ++empty;
}
```

Is there potential for trouble here?

1.1 Critical Section Usage Model

(for n processes, $1 \leq i \leq n$)

```
Pi:
do {
   mutexbegin();       /* CS entry */
   CSi;
   mutexend();         /* CS exit  */
   non-CS
} while (!done);
```

1.2 A Simple, Primitive Hardware Solution

1. Just disable interrupts.
2. Umm, what about user processes?
3. Why this doesn't work with multiprocessors.
4. This is dangerous.

2 Cooperating Processes

1. Must cooperating processes synchronize under all conditions? (Don't forget single writer performing atomic writes/multiple readers.)
2. What does *atomic* mean?
3. Recall necessary and sufficient conditions: Mutual exclusion, progress, and bounded waiting.

3 A Hardware Solution: TAS Instruction

TAS: Test And Set. Semantics:

```
int TAS(int& val)
{
   int temp;

   temp = val;   // Body performed atomically.
   val = 1;
   return temp;
}
```

A partial solution to the critical section problem for n processes:

```
// Initialization
int lock = 0;

void MutexBegin()
{
   while (TAS(lock))   // Ugh.  A spin lock.
      ;
}


void MutexEnd()
{
   lock = 0;
}
```

Prove that this is a solution to the C. S. problem.

4 Semaphores

1. Created by Dijkstra (Dutch)
2. A semaphore is an integer flag, indicating that it is safe to proceed.
3. Two operations:
   (a) Wait (p) — *proberen*, test:

   ```
   wait(s) {
      while (s == 0)
         ;
      s--;
   }
   ```

   Test and (possible) decrement executed atomically (usually achieved through hardware means).

   (b) Signal (v) — *verhogen*, increment:

```
signal(s) {
   s++;
}
```

   (c) Why not resort to hardware methods?

4. These are operations provided by the kernel. Wait and signal are atomic operations.

4.1 Critical Section Problem Solution

1. Critical section solution:

```
semaphore mutex = 1;

mutexbegin: wait(mutex);
mutexend:   signal(mutex);
```

   (a) Mutual exclusion is achieved: consider a contradiction.

   (b) Progress is achieved: *someone* got the semaphore.

   (c) Bounded waiting depends on how the wait queue is implemented (if at all).

4.2 Usage Examples

1. Interrupt signalling:

```
semaphore sig = 0;

int_hndl:
signal(sig);

driver:
startread();
wait(sig);
```

2. Process synchronization:

   ```
   semaphore flag = 0;


   process1()
   {
      p1Part1();   // This will complete before p2part2() begins.
      signal(flag);
      p1Part2();
   }


   process2()
   {
      p2part1();
      wait(flag);
      p2part2();
   ```

3. Resource management (pool of buffers)

   Producer/Consumer problem:

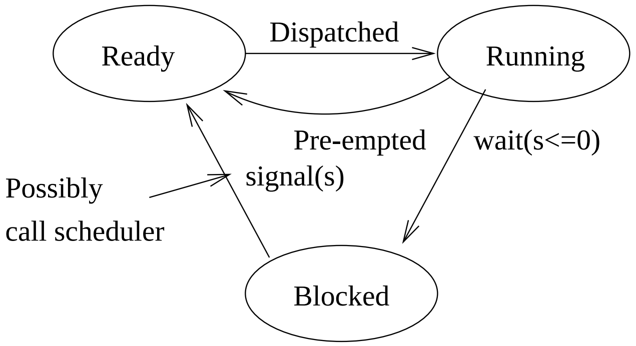
   ```
   semaphore count = N;
   semaphore mutex = 1;

   getbuf:
   wait(count);               /* order important here */
   wait(mutex);
   <grab unallocated buffer>
   signal(mutex);
   return(buffer);

   relbuf:
   wait(mutex);
   <release buffer>
   signal(mutex);
   signal(count);
   ```

4.3 A Better Semaphore

1. Above semaphores inefficient — spinlocks. Let waits which cause busy waits actually block the process:



Associate a “blocked” queue with each semaphore.

```
typedef struct semaphore {
   int   value;
   pcb   *head;
}
```

Semaphore creation:

```
semaphore *createsem(int value) {

   semaphore   *sem;

   sem = get_next_sem();
   sem->value = value;
   sem->head = NULL;
   return (sem);
}

void wait(semaphore *sem) {        /* need mutex goo here */

   if (--sem->value < 0) {
      <update status of current process>
      insqu(sem->head->prev, current);
      scheduler();
   }
}

void signal(semaphore *sem) {         /* mutex */

   pcb   *proc;
```

```
   if (++sem->value <= 0) {
      proc = remqu(sem->head->next);
      <update status of proc>
      ordinsqu(ready, proc);
      if (proc->prio > current->prio)
         scheduler();
   }
}
```