CMSC 412
Fall 2004

Implementing Synchronization

Review

- What is synchronization and why do we need it?
- What is deadlock?
- How are synchronization and deadlock related?

- Reading: chapter 7.

Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its (related) critical section.

Implementing Critical-Section

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the *n* processes.

Assumptions

- Some instructions are atomic
  - load, store, test instructions cannot be interrupted
- Hardware configuration can vary
  - Single or multiple processors

Simpler Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

do {
  entry section
    critical section
  exit section
    remainder section
} **while (1);**

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
  - **int turn** (initially **turn = 0)**
  - **turn - i** $\Rightarrow$ P_i can enter its critical section
- Process P_i

do {
  while (turn != i) ;
  critical section
  turn = j;
  remainder section
} **while (1);**

Algorithm 1

- Satisfies mutual exclusion but not progress.
  - Processes are forced to enter their critical sections alternately.
  - One process not in its critical section thus prevents the other from entering its critical section.

Algorithm 2

- Shared variables
  - **boolean flag[2];**
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** ⇒ P_i ready to enter CS
- Process P_i

```
do {
  flag[i] := true;
  while (flag[j]) ;
  critical section
  flag [i] = false;
  remainder section
} while (1);
```

Algorithm 2

- Satisfies mutual exclusion, but not progress requirement.
  - Both processes can end up setting their flag[] variable to true, and thus neither process enters its critical section!

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
do {
   flag [i]:= true;
   turn = j;
   while (flag [j] and turn = j) ;
   critical section
   flag [i] = false;
   remainder section
} while (1);
```

Algorithm 3

- Meets all three requirements; solves the critical-section problem for two processes.
  - One process is always guaranteed to get into its critical section.
  - Processes are forced to take turns when they both want to get in.

Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm

- Notation <≡ lexicographical order (ticket #, process id #)
  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0,\ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for i - 0, …, n – 1
- Shared data

  boolean choosing[n];

  int num[n];

  Data structures are initialized to **false** and **0** respectively

Bakery Algorithm

```
do {
   choosing[i] = true;
   num[i] = max(num[0], num[1], …, num [n - 1])+1;
   choosing[i] = false;
   for (j = 0; j < n; j++) {
     while (choosing[j]) ;
     while ((num[j] != 0) && (num[j],j) < (num[i],i)) ;
   }
   critical section
   num[i] = 0;
   remainder section
} while (1);
```

Synchronization Hardware

- Uniprocessor machine
  - Disable/enable interrupts:

Process P_i

```
do {
        while (!Interrupts_Enabled()) ;
        Disable_Interrupts()
        critical section
        Enable_Interrupts()
        remainder section
}
```

Disabling Interrupts

- Doesn't work for multiprocessors
- Doesn't permit different groups of critical sections

Synchronization Hardware

- Test-and-Set (Tset):
  - Test and modify the content of a word atomically

```
boolean TestAndSet(boolean *target) {
  boolean rv = *target;
  *target = true;
  return rv;
}
```

Mutual Exclusion with TSet

- Shared data:

```
boolean lock = false;
```

- Process P_i

```
do {
  while (TestAndSet(lock)) ;
  critical section
  lock = false;
  remainder section
}
```


10

Synchronization Hardware

- Swap
  - Atomically swap two variables.

```
void Swap(boolean *a, boolean *b) {
  boolean temp = *a;
  *a = *b;
  *b = temp;
}
```

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

```
boolean lock;
boolean waiting[n];
```

- Process P_i

```
do {
  key = true;
  while (key == true) Swap(lock,key);
  critical section
  lock = false;
  remainder section
}
```

Semaphores

- Synchronization tool
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations:

wait (*S*):

while $S \leq 0$ **do** ***no-op*;**

***S*--;**

signal (*S*):

***S*++;**

Critical Section of *n* Processes

- Shared data:

semaphore mutex; //initially *mutex* = 1

- Process *Pi:*

```
do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while (1);
```

Semaphore Implementation

- To avoid busy-waiting, define a semaphore as a record

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
```

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(*P*)** resumes the execution of a blocked process **P**.

Implementation

- Semaphore operations now defined as

wait(*S*):
```
S.value--;
if (S.value < 0) {
   add this process to S.L;
   block();
}
```

signal(*S*):
```
S.value++;
if (S.value <= 0) {
   remove a process P from S.L;
   wakeup(P);
}
```

Two Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.
- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.
- Can implement a counting semaphore *S* as a binary semaphore.

Implementing S as a Binary Semaphore

Data structures:

binary-semaphore S1, S2;

int C:

Initialization:

S1 = 1

S2 = 0

C = initial value of semaphore S

Implementing *S* (wait)

```
wait(S1);
C--;
if (C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);
```

Implementing *S* (signal)

```
wait(S1);
C ++;
if (C <= 0)
    signal(S2);
else
    signal(S1);
```

Semaphores for Ordering

- Execute B in P_j after A executed in P_i
- Use semaphore flag initialized to 0
- Code:

P_i	P_j
⋮	⋮
A	wait(flag)
signal(flag)	B

Classical Synchronization Problems

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1
  *where **n** is the size of the buffer.*

Producer Process

```
do {
        … produce an item in nextp
        wait(empty);
        wait(mutex);
        … add nextp to buffer
        signal(mutex);
        signal(full);
} while (1);
```

Consumer Process

```
do {
        wait(full)
        wait(mutex);
        … remove an item from buffer
        signal(mutex);
        signal(empty);
        … consume removed item
} while (1);
```

Readers-Writers Problem

- Shared data

  semaphore mutex, wsem;

  Initially

  mutex = 1, wsem = 1, readcount = 0

18

Writer Process

```
wait(wsem);
… writing is performed
signal(wsem);
```

Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
   wait(wsem);
signal(mutex);
… reading is performed
wait(mutex);
readcount--;
if (readcount == 0)
   signal(wsem);
signal(mutex):
```

Comments

- Readers have priority - a writer can gain access to the data only if there are no readers (i.e. when readcount is zero, signal(wsem) executes)
- Possibility of starvation - writers may never gain access to data

Reader Process

```
wait(rsem);
wait(x);
 readcount++;
 if (readcount == 1) then
                  wait(wsem);
signal(x);
signal(rsem);
```

Read here

```
wait(x);
 readcount--;
 if (readcount == 0) then
                   signal(wsem)
signal(x)
```

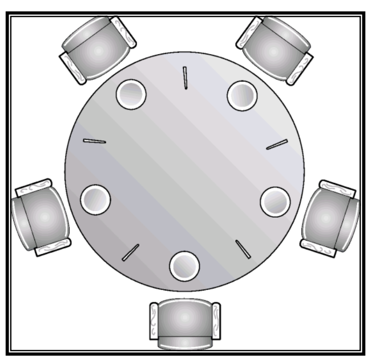
Writer Process

```
wait(y);
    writecount++:
    if writecount == 1 then
                      wait(rsem);
signal(y);
wait(wsem);
```

Write here

```
signal(wsem);
wait(y);
    writecount--;
    if (writecount == 0) then
                     signal(rsem);
signal(y);
```

Dining-Philosophers Problem



- Shared data

  semaphore chopstick[5];

Initially all values are 1

Dining-Philosophers Problem

- Philosopher *i*:

```
do {
  wait(chopstick[i])
  wait(chopstick[(i+1) % 5])
  … eat
  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);
  … think
  } while (1);
```

Comments

- This solution can deadlock
  - Imagine each philosopher grabbing one chopstick (wait()), and then being context-switched. None can make progress
  - Can you draw a wait-graph to show this situation?