Administrivia

- Project progress/help?
- Homework on class webpage by tonight; due Tue. 26
- Project 2 will be assigned on Thursday
- Need to schedule first exam (how about Th. 3/6?)
- Last Time:
  - Concurrency software algorithms
  - Test and set
Lets Set the Scene Again

- Multiple threads that share data (remember the add-one example)

- Cannot make any assumptions about the scheduler: a switch to a different thread can happen after any assembly language instruction

- Critical section problem. We want: (1) mutual exclusion; (2) progress; (3) no starvation
Third Attempt: Announce Interest Early

bool flag[2];

Process 0          Process 1
.
.
flag[0] = 1;        flag[1] = 1;
while (flag[1] != 0); while (flag[0] != 0);
[Critical Section]  [Critical Section]
flag[0] = 0;        flag[1] = 0;

• Problem Solved?
  – Only one process can enter critical region at a time

• Is starvation a problem?

• Still a wrong Solution! why?
Fourth Attempt: Double check and Back-off

bool flag[2];

Process 0
.
.
flag[0] = 1;
while (flag[1] != 0) {
    flag[0] = 0;
    wait a short time
    flag[0] = 1;
}

[Critical Section]
flag[0] = 0;

Process 1
.
.
flag[1] = 1;
while (flag[0] != 0) {
    flag[1] = 0;
    wait a short time
    flag[1] = 1;
}

[Critical Section]
flag[1] = 0;

• Finally a correct implementation?
Dekker’s Algorithm

```c
bool flag[2];
int turn = 0;

Process 0
.
.
flag[0] = 1;
while (flag[1] != 0)
if (turn == 1) {
    flag[0] = 0;
    while (turn == 1);
    flag[0] = 1;
} /*if*/
/*while*/
[Critical Section]
flag[0] = 0;
turn = 1;

Process 1
.
.
flag[1] = 1;
while (flag[0] != 0)
if (turn == 0) {
    flag[1] = 0;
    while (turn == 0);
    flag[1] = 1;
} /*if*/
/*while*/
[Critical Section]
flag[1] = 0;
turn = 0;
```

- The two flags solve the mutual exclusion problem; use the turn (as per the first implementation) to solve simultaneous interest problem
- Do we have the alternating execution problem?
More Elegant Solution: Peterson’s Algorithm

```c
bool flag[2];
int turn = 0;

Process 0
.
.
flag[0] = 1;
turn = 1;
while (flag[1] == 1 && turn == 1);

[Critical Section]
flag[0] = 0;

Process 1
.
.
flag[1] = 1;
turn = 0;
while (flag[0] == 1 && turn == 0);

[Critical Section]
flag[1] = 0;
```

- Does this work? How?
- Is it fair (starvation/alternating execution?)
- How can we prove its correctness?
Bakery Algorithm

//choosing, ticket are shared
...
choosing[i] = TRUE;
ticket[i] = max (ticket[0], ticket [1] ...
    ticket [n]) + 1;
choosing[i] = FALSE;
for(j = 0; j < n; j++) {
    while (choosing[j] == TRUE);
    while (ticket[j] != 0 &&
        (ticket[j],j) < (ticket [i],i));
}
  [Critical Section]
ticket[i] = 0;
  ...

• (ticket[j],j) < (ticket[i],i) refers to the comparison including using the process number as tie-breaker if tickets equal

• Take your time, think about it

• Does it satisfy the three requirements?
Test and Set

```c
bool lock = 0;
Process 0               Process 1
    .                      .
    .                      .
while (testAndSet(lock)); while (testAndSet(lock));
[Critical Section]      [Critical Section]
lock = 0;               lock = 0;
```

- One Instruction that tests a shared variable and sets it to 1 **atomically**
- Simpler but still busy waits
- Generalizes to any number of processes/locks
- What are the implications if used on a Shared Memory Multiprocessor?
- Is waiting bounded?
- Example of **test-and-op** class of primitives
Test and Set for $n$ Processes with Bounded Wait

waiting[i] = 1;
key[i]=1;
while(waiting[i] && key[i])
    key[i] = testAndSet(lock);
waiting[i] = 0;

[Critical Section]

j = i+1 % n
while ((j != i) && !waiting[j])
    j = j + 1 % n;
if (j == i)
    lock = 0;
else
    waiting[j] = 0;
Busy waiting vs. Blocking

• All the methods discussed so far employ busy waiting
  – Such locks are called spin locks
    * A process waiting on a lock keeps spinning its wheels wasting CPU time

• Idea: use a blocking lock and signalling for a more efficient implementation – what is the tradeoff?

• Are there situations where spin locks are more efficient than blocking locks?

• Use locks as low-level primitives, but do not busy wait

• Semaphores (Dijkstra) is a widely used locking mechanism that uses this idea
Semaphores

- Informally, multi-valued, queued, blocking lock
- Semaphore consists of a value and a queue
- Acquiring a Semaphore (aka, `wait`, `down`, or `P`):
  - If value is positive, decrement it and let the process in
  - If value is zero, put the process at the end of the queue
- A process *releases* a semaphore (aka, `signal`, `up`, or `v`)
  - If queue is not empty, head of the queue is allowed to acquire S
  - If the queue is empty, increment S’s value
- Access to the semaphore information should be atomic (how?/why?)
Mutual Exclusion Using Semaphores

Semaphore mySem; //initialized to 1
...
wait(mySem);

[Critical Region]

signal(mySem);
...

- Semaphore operations must be atomic; how do we implement them?
typedef struct sem {
    int value;
    Queue q;
} Semaphore;

void wait(Semaphore S) {
    if (S.value == 0)
        Add to queue and block  // proceed when unblocked
    else
        S.value--;
}
void signal (Semaphore S) {
    if (!empty(S.q))
        Unblock Top of queue
    else
        S.value++;
}

• Is there a problem in this implementation?
 Implementation

void wait(Semaphore S) {
  Lock(S.Lock); // added Lock variable to Semaphore
  if (S.value == 0)
    Add to queue and block; // proceed when unblocked
  else
    S.value--;
  Unlock(S.Lock);
}

void signal (Semaphore S) {
  Lock(S.Lock);
  if (!empty(S.q))
    Unblock Top of queue;
  else
    S.value++;
  Unlock(S.Lock);
}

• Lock and Unlock can be any of our software locks, or even disabling interrupts

• Is there a problem in this implementation? think about what happens when a process blocks
More Elaborate Synchronization

Semaphore sem1, sem2; // initialized to 0

P0                     P1
...                    ...
wait(sem1);            [Activity B]
[Activity A]            signal(sem1);
signal(sem2);          wait(sem2);
...                    ...
[Activity C]
...                    ...

• You will be doing this kind of stuff for your NachOS assignment
Deadlock and Starvation

- Deadlock happens when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes (circular dependency)

- S and Q are two semaphores initialized to 1:

  P0
  ...
  wait(S);
  wait(Q);
  ...
  signal(Q);
  Signal(S);

  P1
  ...
  wait(Q);
  wait(S);
  ...
  Signal(S);
  Signal(Q);

- Indefinite blocking: a process may never be removed from the semaphore queue (while others are)
Binary vs Counting Semaphore

- There are two types of semaphores
  - Counting semaphore – can take any integer value (what we have discussed so far)
  - Binary semaphore – value can be only 0 or 1; can be simpler to implement

- We can implement a counting semaphore using binary semaphores; how?
Implementation

- **Wait:**

  ```
  //S1, S3 initialized to 1; S2 to 0
  wait(S1);
  wait(S3);
  value--;
  if(value < 0) {
    signal(S1);
    wait(S2);
  } else signal(S1);
  signal(S3);
  ```

- **Signal:**

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

- **S1** – mutual exclusion; S3 and S2 enforce the count
Revisiting Producer Consumer

//Binary semaphore Mutex, initialized to 1
//Binary semaphore empty, initialized to 0
int n = 0;

// Producer
while (1) {
    produce;
    wait(mutex);
    append();
    n++;
    if(n == 1)
        signal(empty);
    signal(mutex);
}

//consumer
wait (empty);
while(1) {
    wait(mutex);
    take();
    n--;
    signal(mutex);
    consume;
    if(n == 0)
        wait(empty);
}

• Wrong Implementation

• Can you see the flaw in this program? Its tricky.
Can you fix it?
Correct Implementation Using Binary Semaphores

//Binary semaphore Mutex, initialized to 1
//empty, initialized to 0
int n = 0, m;

// Producer
while (1) {
produce()
wait(mutex);
append();
n++;
if(n == 1)
    signal(empty);
signal(mutex);
}

//consumer
wait(empty);
while(1) {
wait(mutex);
take();
n--;
m = n;
signal(mutex);
consume();
if(m == 0)
    wait(empty);
}

• Why does this additional assignment solve anything?

• A little clumsy
Counting Semaphore Implementation

//Semaphore mutex = 1, Semaphore count = 0

// Producer
while (1) {
    produce()
    wait(mutex);
    append();
    signal(mutex);
    signal(count);
}

// Consumer
while(1) {
    wait(count);
    wait(mutex);
    take();
    signal(mutex);
    consume()
}

• Is this correct?
• Why not move the produce/consume before the Release of mutex?
More Elegant Solution

//Initial values: mutex = 1, full = 0, empty = n
Producer:
while (1) {
    produce an item in nextp;
    wait(empty);
    wait(mutex);
    buffer.append(nextp);
    signal(mutex);
    signal(full);
}

Consumer:
while(1) {
    wait(full);
    wait(mutex);
    nextc = buffer.nextItem();
    signal(mutex);
    signal(EMPTY);
}
Semaphore Discussion

- A more powerful and efficient mechanism for locking
- Activities interfere with each other only if they access the same semaphore
- No busy wait (or is there?)
- As with everything discussed so far, relies on well behaved processes to release the lock/semaphore on their way out
- A semaphore is a non-preemptable resource — subject to deadlock
Readers/Writers Problems

• So far, we have assumed that all accesses must be mutually exclusive

• With Readers/Writers problems
  – Any number of readers may access the resource
  – Only one writer at a time may access it
  – No readers should be reading when a writer is writing

• When is mutual exclusion necessary?

• Why not just use full mutual exclusion?

• Can you think of problems that are in this class?
Solution to Readers Writers Problem

//Semaphore mutex initialized to 1
//Semaphore write initialized to 1
readers writers

... ...
wait(mutex); wait(write);
readers++; wait(write);
if(readers == 1) WRITE;
    wait(write);
signal(mutex); signal(write);
READ;

wait(mutex);
readers--; signal(write);
if(readers == 0) signal(write);
signal(mutex);

• Problems?
Bonus HW Problem: Writers Have Priority

//All semaphores initialized to 1
reader

... wait(one_reader);
wait(read);
wait(mutex1);
readers++;
if(readers == 1)
  wait(write);
signal(mutex1);
signal(read);
signal(one_reader);
READ;
wait(mutex1);
readers--;
if(readers==0)
  signal(write);
signal(mutex1);

writer

... wait(mutex2);
writers++;
if(writers == 1)
  wait(read);
signal(mutex2);
wait(write);
WRITE;
signal(write);
wait(mutex2);
writers--;
if(writers == 0)
  signal(read);
signal(mutex2);