Third Attempt: Announce Interest Early

bool flag[2];

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

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

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

[Critical Section]
```
flag[0] = 0; flag[1] = 0;
turn = 1; 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; flag[1] = 1;
turn = 1; turn = 0;
while (flag[1] == 1 && turn == 1);
```

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

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

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

Bakery Algorithm

```c
//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));
} /* while */
[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
```
while (testAndSet(lock));
```

Process 1
```
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

\[
\begin{align*}
\text{waiting}[i] &= 1; \\
\text{key}[i] &= 1; \\
\text{while}(\text{waiting}[i] \&\& \text{key}[i]) \\
&\quad \text{key}[i] = \text{testAndSet}(\text{lock}); \\
&\quad \text{waiting}[i] = 0;
\end{align*}
\]

[Critical Section]

\[
\begin{align*}
\text{if } i &= 1 \mod n \\
\text{while } ((j != i) \&\& \!\! \text{waiting}[j]) \\
&\quad j = j + 1 \mod n; \\
\text{if } (j == i) \\
&\quad \text{lock} = 0; \\
\text{else} \\
&\quad \text{waiting}[j] = 0;
\end{align*}
\]

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, \texttt{wait}, \texttt{down}, or \texttt{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 \textit{releases} a semaphore (aka, \texttt{signal}, \texttt{up}, or \texttt{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 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?

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

• 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);
Signal(Q);

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

- 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:

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

signal(S3);
```

- Signal:

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

S1 – mutual exclusion; S3 and S2 enforce the count

Revisiting Producer Consumer

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

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

- Wrong Implementation
- Can you see the flaw in this program? Its tricky. Can you fix it?

Correct Implementation Using Binary Semaphores

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

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

- Why does this additional assignment solve anything?
- A little clumsy
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

Counting Semaphore Implementation

//Semaphore mutex = 1, Semaphore count = 0

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

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

- 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();
    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);
}

Readers/Writers Problems

- So far, we have assumed that all accesses must be mutually exclusive
- Readers should be able to read at the same time
- When is mutual exclusion necessary?
- Why not just use full exclusion?
- With Readers/ Writers problems
  - Any number of readers may access it
  - Only one writer at a time may access it
  - No readers should be reading when a writer is writing

- 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 
wait (mutex);
readers++; 
if(readers == 1) 
wait (write);
signal (mutex); 
readers--; 
if(readers == 0)
signal (mutex);

• Problems?

Bonus HW Problem: Writers Have Priority

//All semaphores initialized to 1

readers 
writters

wait (mutex);
wait (write);
writters++;
if(readers == 1)
wait (write);
signal (mutex);
readers--; 
if(readers == 0) 
signal (mutex);

wait(mutex2);
write;
signal(mutex2);
wait(mutex1);
readers++; 
if(readers == 1) 
wait (write);
signal (mutex1);
write;
signal (one_reader);
READ;
wait(mutex1);
readers--; 
if(readers == 0)
signal (mutex);

wait(mutex2);
write;
signal (one_reader);
READ;
wait(mutex2);
readers--; 
if(readers == 0)
signal (mutex2);

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

wait(mutex2);
write;
signal (read);
signal (mutex2);