ELEC 377 – Operating Systems

Week 3 – Class 1

Reminders

- 2nd session of Lab 1 today
  - Write up due 4PM Sept 27th
- Quiz #1 tomorrow

Last Class

- Synchronization
- Critical Sections and Race Conditions
- Criteria for Solutions
- 2 Process Solution - 3 Algorithms

Today

- Synchronization
- Bakery Algorithm
- Hardware Support
- Classic Problems

Critical Sections - General Model

```
do {
    entry section

                critical section

    exit section

                remainder section

} while (1);
```

n Processes - Bakery Algorithm

- Not in V8 of Textbook, but we are covering it anyways
- Based on pick a number in Bakery, Deli’s, Government offices.
- Pick the next number (smallest number goes first)

- **Problem:** picking the number
  - ◊ real world physical number ticket – only one!!
- Race conditions in picking numbers, but the numbers are monotonic increasing (1,2,3,3,4,5,5,…)
- numbers not always unique
  - ◊ tie goes process with lowest PID.

n Processes - Bakery Algorithm

```
do {
  choosing[i] = true;
  num[i] = max(num[0],..,num[n]) + 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);
```

Bakery Algorithm - Choosing

```
choosing[i] = true;
num[i] = max(num[0],..,num[n]) + 1;
choosing[i] = false;
```

- Remember: i is the current process
- choosing for us is true when picking a number
- max function and addition **not** *atomic*
- interrupts *can* happen here

Bakery Algorithm - Choosing

```
choosing[i] = true;
num[i] = max(num[0],..,num[n]) + 1;
choosing[i] = false;
```

- What are we guaranteeing? (Case analysis)

Case 1:

process j is outside of critical section, outside of entry routine (i.e. in its remainder section)

- num[j] = 0, choosing[j] = false
- if process j enters after us, then they will have a higher ticket number than us

Bakery Algorithm - Choosing

```
choosing[i] = true;
num[i] = max(num[0],..,num[n]) + 1;
choosing[i] = false;
```

- What are we guaranteeing? (Case analysis)

Case 2:

process j is in critical section ahead of us

☐ num[j] ≠ 0, num[j] < num[i], choosing[j] = false

☐ when they leave the critical section, num[j] = 0

Bakery Algorithm - Choosing

```
choosing[i] = true;
num[i] = max(num[0],..,num[n]) + 1;
choosing[i] = false;
```

- What are we guaranteeing? (Case analysis)

Case 3:

process j has completed choosing before us, has lower number

☐ num[j] ≠ 0, num[j] < num[i], choosing[j] = false

☐ they will go ahead of us into the critical section

☐ when they leave the critical section, num[j] = 0

Bakery Algorithm - Choosing

```
choosing[i] = true;
num[i] = max(num[0],..,num[n]) + 1;
choosing[i] = false;
```

- What are we guaranteeing? (Case analysis)

Case 4:

process j has completed choosing after us, has higher number

☐ num[j] ≠ 0, num[j] > num[i], choosing[j] = false

☐ we go ahead of them, they wait for us as case 3

Bakery Algorithm - Choosing

```
choosing[i] = true;
num[i] = max(num[0],..,num[n]) + 1;
choosing[i] = false;
```

- What are we guaranteeing? (Case analysis)

Case 5:

both our process and process j are choosing a number at the same time, both finished

☐ num[j] ≠ 0, num[j] = num[i], choosing[j] = false

☐ lowest process goes ahead

Bakery Algorithm - Choosing

```
choosing[i] = true;
num[i] = max(num[0],..,num[n]) + 1;
choosing[i] = false;
```

- What are we guaranteeing? (Case analysis)

Case 6:

process j is still choosing, so we don’t know what the ticket number for j is.

☐ It might be lower (interrupt happened after j chose a number, but before we chose a number)

☐ It might be higher

☐ choosing[j] = true

Only case where we are unsure

Bakery Algorithm - Choosing

```
choosing[i] = true;
num[i] = max(num[0],..,num[n]) + 1;
choosing[i] = false;
```

- Looking at other ticket numbers is **not** *atomic*
- So when we go to look at other processes’ ticket numbers, we first check to see if the number is *stable*

◊ choosing[j] = false;

◊ once choosing[j] = false, then their ticket number can never be *lower*, it can only **increase.**

◊ If it changes, it **must** be greater than our ticket number

n Processes - Bakery Algorithm

```
for (j = 0; j < n; j++){
  while(choosing[j]);
  while(num[j] != 0 &&
        ((num[j],j)<(num[i],i))); //empty
}                                          // loop
```

- Look at each other process in turn (not atomic)
- Check each process in process id order (lowest process id first)
- wait for them to choose, then check ticket number.
- Only go to next j if we are ahead of the current j.
- When we hit the end of the loop, we must be at the front of the list

n Processes - Bakery Algorithm

```
for (j = 0; j < n; j++){
  while(choosing[j]);
  while(num[j] != 0 &&
        ((num[j],j)<(num[i],i)));
}
```

- Look at each other process in turn (not atomic)
- Check each process in process id order (lowest process id first)
- wait for them to choose, then check ticket number.
- Only go to next j if we are ahead of the current j.
- When we hit the end of the loop, we must be at the front of the list

Today

- Synchronization
- Bakery Algorithm
- Hardware Support <<<<<<<
- Classic Problems

Hardware Support

- Some hardware provides support for synchronization
- ◊ *atomic* instructions
- ◊ cannot be interrupted
- ◊ read-modify-write
- ◊ single processor/multi processor

- Test and Set
- ◊ read a boolean variable
- ◊ set the boolean variable to true
- ◊ atomic, if another process reads after this instruction starts, then the variable will be *true*. Only one process can read the value *false*.

Test And Set – Use

```
do {
  while(TestAndSet(lock));

    critical section

  lock = false;

     remainder section

} while(1);
```

- Bounded wait not satisfied

Hardware Support

- Swap
  - ◊ exchange a value between a register and memory
  - ◊ atomic

```
do {
  key = true;
  while(key == true) { Swap(lock,key); }
    critical section
  lock = false;
     remainder section
} while(1);
```

- use swap to implement test and set
- Bounded wait still not satisfied

Test And Set – Bounded Wait

```
do {
  waiting[i] = true;
    key = true;
    while(waiting[i] && key)
    key = TestAndSet(lock);
  waiting[i] = false
    critical section
  j = (i+1) % n
  while(j!=i && !waiting[j]) j=(j+i)%n;
  if(i == j) lock = false
  else waiting[j] = false;
     remainder section
} while(1);
```

Test And Set – Bounded Wait

- pass the key approach
- When exiting the critical section, don’t release the lock, pass the lock to someone who is already waiting
- pass the lock in increasing process id order (with wrap around)
- if one process leaves critical section and gets back to entry point before a context switch, must wait for all other processes that are waiting
- If waiting, each other process may execute critical section at most once

Bounded Wait - Entry

```
waiting[i] = true;
  key = true;
  while(waiting[i] && key)
  key = TestAndSet(lock);
waiting[i] = false
```

- waiting flags (initially all false)
- lock initially false
- Process i indicates waiting with wait flag
- Once in the critical section no longer waiting
- loop on waiting flag and on lock
- Enter critical section if we get the lock or if it is passed to us - key may never be false for us

Bounded Wait - Exit

```
j = (i+1) % n
while(j!=i && !waiting[j]) j=(j+i)%n;
if(i == j) lock = false
else waiting[j] = false;
```

- We don’t immediately set the lock to false (we don’t release the lock).
- Instead we pass the key to the next waiting process (in process id order).
- Iterate through processes looking for processes with waiting flag true. If we reach ourselves, no processes waiting.

Today

- Synchronization
- Hardware Support
- Semaphores <<<<<<<<
- Classic Problems

Semaphores

- All of the solutions to date do not generalize easily
- Busy Waiting - waste of CPU cycles (spinlock)
- General Solution - Semaphore

◊ integer variable
◊ two atomic operations (wait and signal)

```
wait(S)                                   P(S)
    while (S ≤ 0);
    S--;

signal(S)                                 V(S)
    S++;
```

Semaphore - Critical Sections

```
semaphore mutex = 1



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

Semaphore - Blocking Solution

```
typedef struct {
    int value;
    struct process * L;
} semaphore;
wait(S):
  S.value--;
  if (S.value < 0){
    add process to S.L
    block();}
signal(S):
  S.value ++;
  if (S.value <= 0) {
    get P from S.L
    wakeup(P) }
```

Semaphores

- Semaphores generalize easily
- Can make one line wait for another.
- Must be careful of deadlock and starvation

◊ deadlock

```
wait(S)          wait(T)
wait(T)          wait(S)
…                …
signal(S)        signal(T)
signal(T)        signal(S)
```

◊ starvation – signal is never made, process never wakes up

Two Types of Semaphores

- Counting Semaphores
- Binary Semaphores
- Must be careful of deadlock and starvation

◊ Simpler
can used two binary semaphores to implement a counting semaphore.

Today

- Synchronization
- Hardware Support
- Semaphores
- Classic Problems <<<<<<<<

Bounded Buffer

- Shared Data:

  semaphore full, empty, mutex;

  initially:

  full = 0; empty = n; mutex = 1;

  n is size of buffer;

Bounded Buffer - Producer

```
wait(empty);
wait(mutex);
  ... add to buffer ...
signal(mutex);
signal(full);
```

- Note that mutex is symmetric, empty and full semaphores are not

Bounded Buffer - Consumer

```
wait(full);
wait(mutex);
  ... remove from buffer ...
signal(mutex);
signal(empty);
```

- the empty semaphore contains the number of empty spaces left in the buffer
- the full semaphore contains the number of items in the buffer