Synchronization

CS 537 - Introduction to Operating Systems

What is Synchronization?

- Recall that there is no guarantee about the ordering of instructions between processes (or threads)
- Synchronization is providing explicit control about the ordering of operations

Machine Level Instructions

- Single high level language (C, Java, etc.) are often broken down into multiple machine instructions
- Example

```
...                        ld   r1, [count]
count ++;      ===>        add r1, 1
...                        st   [count], r1
```

- Interrupt or context switch can occur between any of the above instructions
- Most high level instructions are not atomic

Atomocity

- Everything happens *at once*
- Machine instructions are atomic
  - ld r1, [count]
  - above instruction can not be broken up by interrupt
- High level instructions are not atomic
  - count++;
  - this is actually 3 machine level instructions
  - an interrupt can occur in the "middle" of instr.

Producer-Consumer Revisitted

- Let us consider a small section of code

Producer Thread	Consumer Thread
. . .	. . .
buffer[in] = objProduce;	objConsume = buffer[out];
count++;	count--;
. . .	. . .

- Remember that count++ (count--) is actually 3 instructions
- One possible interleavening of producer and consumer

prod	ld r1, [count]
cons	ld r2, [count]
prod	add r1, 1
cons	sub r2, 1
prod	st [count], r1
cons	st [count], r2

  - for above ordering, value of count is 2
- Depending on ordering, could be 2, 3, or 4

Race Condition

- Previous example is an example of a "Race Condition"
  - two threads "race" to place a value in memory
  - no way to know which one will "win"
- Very bad bug
  - difficult to duplicate because ordering may be different from one run to another
  - without consistent output, hard to find bugs
  - producer-consumer example may run fine as long as count stays between 2 and 9

Critical Section

- If multiple threads with access to shared data that is writeable, then access to the data by each thread must be controlled
- The piece of controlled data for each thread is called its *critical section*
- Banker example
  - one account for 2 people (Jane and John Doe)
  - 2 different bank tellers

Banking Example

- The Doe's current balance is $1000 (B = $1000)
- John deposits $100 with teller 1
- Jane deposits $100 with teller 2
- Teller 1 reads current balance (B = $1000)
- Teller 2 reads reads balance (B = $1000)
- Teller 1 adds John's deposit to balance (B = $1100)
- Teller 2 adds Jane's deposit to balance (B = $1100)
- $100 dollars was lost
- Need to control access by tellers to deposits
  - one teller can't read balance while another is doing a transaction

Banking Example

```
double balance;

void deposit(double amount) {
    enterCriticalSection();
    balance += amount;
    leaveCriticalSection();
}

int main() {
    balance = atoi(argv[1]);
    createMultipleThreads();  // creates multiple threads that call deposit
    waitForThreads();   // wait for threads to finish

    return 0;
}
```

Banking Example

- So what are the *enterCriticalSection* and *leaveCriticalSection* functions
- Two basic requirements for correctly protecting critical section
  - mutual exclusion: only one thread in critical section at a time
  - progress: if no thread in critical section a thread can enter without waiting

Protection Algorithm 1

```
int turn;   // initialized to zero in main()

void enterCriticalSection(int id) {
    while(turn != id)
            yield();
}

void leaveCriticalSection(int id) {
    turn = 1 – id;
}
```

Protection Algorithm 1

- Insures mutual exclusion
- Does NOT guarantee progress
  - imagine thread 0's turn
  - thread 0 is not in the critical section
  - thread 1 cannot enter critical section
    - progress says that it should be able to
  - worst case scenario, thread 0 ends without ever entering (or leaving) critical section
    - it will never be thread 1's turn (thread 1 will never advance any further)

Protection Algorithm 2

```
int flag[2];   // initialize both flag[0] and flag[1] to 0 in main()

void enterCriticalSection(int id) {
    int other = 1 – id;
    flag[id] = true;
    while(flag[other] == true)
        yield();
}

void leaveCriticalSection(int id) {
    flag[id] = false;
}
```

Protection Algorithm 2

- Again, guarantees mutual exclusion
- Does NOT guarantee progress
  - what if thread 0 sets flag to true and then a context switch
  - thread 1 sets its flag to true and then blocks in while loop because thread 0's flag is true
  - thread 0 will now also block because thread 1's flag is true

Protection Algorithm 3

```
int turn;   // initialize turn to 0 in main()
int flag[2];   // initialize both flag[0] and flag[1] to 0 in main()

void enterCriticalSection(int id) {
    int other = 1 - id;
    flag[id] = true;
    turn = other;      // give the other guy priority (one thread will win)
    while ( (flag[other] == true) && (turn == other) )
        yield();
}

void leaveCriticaSection(int id) {
    flag[id] = false;
}
```

Protection Algorithm 3

- Combination of algorithm 1 and 2
- Provides both mutual exclusion and progress
- Only yeild if BOTH the other thread wants control (its flag is true) and it is the other threads turn
- Even if both threads “race” to set the shared turn variable, one of them will win
  - if both get to while loop at same time, one will go and the other will yield

Semaphores

- Previous algorithms do not scale well to more than 2 processes
- Another solution - SEMAPHORES!
  - very simple concept

Semaphores

- Each semaphore has a value (S)
- Each semaphore has two methods
  - decrement value (P)
  - increment value (V)
- The P method only returns if $S > 0$ upon entry to P method
- If $S \leq 0$ upon entry to P, thread blocks until $S > 0$

Semaphores

- Example

```
int S;   // initialize semaphore to 1 in main()
void P() {
  while(S ≤ 0);
  S--;
}
void V() {
  S++;
}
```

Semaphores

- For a semaphore to work, P and V methods must be atomic
- As written above they are not
  - we will show how to make them atomic later
- Notice, P and V do not return any value
  - simply by returning, they indicate a thread has either obtained or given-up "ownership" of the semaphore

Banking Example Revisited

```
double balance;

void deposit(double amount) {
    P();
    balance += amount;
    V();
}

int main() {
    balance = atoi(argv[1]);
    createMultipleThreads();  // creates multiple threads that call deposit
    waitForThreads();   // wait for threads to finish

    return 0;
}
```

Using Semaphores

- Guarantees both mutual exclusion and progress
- There can be many threads running now and using the Semaphore for synchronization
  - not just 2 threads like previous 3 algorithms
- Problem
  - *busy wait*
    - if semaphore not available, the thread "spins" on the value
    - if single processor, no other thread can do useful work - including thread that "holds" the semaphore
  - solution is for the thread to block instead of spinning

Blocking Semaphores

- If S < 0, process adds itself to waiting list
- If process sets S to a value greater than zero, it selects a process off of the waiting list (if one exists) and "wakes" it
- Waiting list implemented as a linked list
  - use pointer field in PCB

Blocking Semaphores

```
int S;   // initialize S to 1 in main()

void P() {
  S--;
  if(S < 0)
    block();   // adds the current process to the waiting list and blocks it
}

void V() {
  S++;
  if(S ≤ 0) {
    W = remove();   // remove some process from waiting list;
    wakeup(W);   // make W runnable - doesn't necessarily run it next
  }
}
```

Types of Semaphores

- A program can have multiple semaphores
  - one semaphore for each resource to protect
    - memory location is a resource
- Two type of Semaphores
  - binary semaphore
    - semaphore value never greater than 1
  - counting semaphore
    - semaphore value can be any integer over 0
    - used if multiple numbers of a given resource
      - when S = 0, all the resources are used up

Implementing Semaphores

- Remember, entire P and V operation must be atomic
- Use hardware support to implement
  - disable interrupts
    - okay if uniprocessor
    - won't work for multiprocessor system
  - use special hardware instructions
    - test-and-set
    - swap

Test-and-Set Instruction

- Special, atomic memory operation
- Check a single memory location
  - set a register equal to current value of location
  - then set the location equal to some set value
- Very powerful primitive operation

Test-and-Set Instruction

- Assume memory location is either 0 or 1
  - return value of 1 means no one currently "holds" this memory location
  - return value of 0 means another thread currently "has" the memory location
  - either way, calling thread sets the location to 0
    - perfectly legal to set it to zero if it already is zero
  - calling thread then examines the return value to determine if it can enter the critical section
- Operation is atomic - no interrupt during execution of instruction

Test-and-Set Example

```
int S;   // initialize to some value
int lock;   // initialized to 1

void P() {
  while(test_and_set(lock) != 1);
  S--;
  if(S < 0) {
    add current thread to waiting list;
     lock = 1;
  }
  else { lock = 1; }
}

void V() {
  while(test_and_set(lock) != 1);
  if ( S < 0 ) {
     wakeup some process waiting on S
  }
  S++;
  lock = 1;
}
```

Swap Instruction

- Very similar to test-and-set
  - *swap* returns the value currently stored in a memory location
  - sets the location to a value specified by the user
    - this is main difference from test-and-set
- Used in much the same way
  - Example of instruction
    - *swap(reg1, memS);*
    - after instruction, *memS* will have original value of *reg1* and *reg1* will have original value of *memS*
- Thread checks register for specific value before proceeding