Lecture Topics

- Today: Concurrency
  (Stallings, chapter 5.1-5.3, 5.6)
- Next: continued

Announcements

- Self-Study Exercise #4
- Self-Study Exercise #5
- Project #2 (due 2/2)
- Project #3 (due 2/16)
Exam #1

- Thursday, 2/9 during lecture
- 80 minutes, 18% of course grade
- Topics:
  - computer systems overview
  - operating system overview
  - processes and threads
  - concurrency
- Study suggestions on course website

Concurrency

- Multiprogramming: multiple processes within a uniprocessor system
- Multiprocessing: multiple processes within a multiprocessor system
- Multithreading: concurrent (and possibly simultaneous) execution of threads
Problems with Concurrent Execution

- Concurrent tasks (processes or threads) often need to share data in memory or files
- Actions performed by concurrent tasks depend on the order in which their execution is interleaved, which typically is not deterministic
- Must control access to shared data (otherwise data may become corrupted)

Simple Example

- Process P1 and P2 share access to variable A and both use echo()
- Processes can be interrupted anywhere
- Assume P1 is interrupted after cin >> A
- Assume P2 is interrupted after cout << A
- Then the character echoed by P1 will be the one read by P2 (error!)

```c
void echo()
{
    cin >> A;
    cout << A;
}
```
Simple Example (cont)

- Execution trace:
  
  P1: cin >> A     // user enters X  
  P2: cin >> A     // user enters Z  
  P2: cout << A    // displays Z  
  P1: cout << A    // displays Z

- Some execution traces produce correct results, others do not

Example: Threads

- Main thread:
  
  - initializes "Count" to zero  
  - creates five other threads (all the same)  
  - waits for them to finish, displays "Count"

- Each of the five threads increments "Count"  
  100,000 times

- Total should be 500,000 (5 * 100,000)
Example: Threads (cont)

<1 cse410> g++ -WAll -lpthread threads5.c

<2 cse410> a.out
Count: 129630 (should be 500000)

<3 cse410> a.out
Count: 188825 (should be 500000)

<4 cse410> a.out
Count: 134586 (should be 500000)

The execution results are incorrect. Even worse, the results change for every execution!

The problem is that more than one thread is updating variable "Count" at the same time:

- Get pointer to "Count"
- Load register from memory at pointer
- Add 1 to register
- Store register into memory at pointer
Example: Threads (cont)

- On "cse410" (Intel x86_64)
  - Load register from memory at pointer
  - Add 1 to register
  - Store register into memory at pointer

```assembly
mov 0x0(%rip),%eax
add $0x1,%eax
mov %eax,0x0(%rip)
```

Example: Threads (cont)

Five functions executing concurrently:

```c
void* thread_function( void* arg )
{
    for (int n=0; n<NLOOPS; n++)
    {
        Count++; ← Critical section
    }
    return NULL;
}
```
Revised Example:

```c
void* thread_function( void* arg )
{
    for (int n=0; n<NLOOPS; n++)
    {
        sem_wait( &lock );
        Count++;
        sem_post( &lock );
    }
    return NULL;
}
```

Examples:

Source code for both examples available on the course website:

~cse410/Examples/0202/threads5.c
~cse410/Examples/0202/threads6.c
Critical Sections

- When a process executes code that manipulates shared data, we say that the process is in a **critical section** (CS) for that shared data (aka **critical region**)

- The execution of critical sections must be **mutually exclusive**: at any time, only one process is allowed to execute in its critical section (even with multiple CPUs)

The Critical Section Problem

- The **critical section problem**: design a protocol for tasks so that their actions will not depend on the order in which their execution is interleaved (possibly on many processors)

- Each task requests permission to enter its critical section (CS): **entry section**

- Each task signals that it has left its critical section (CS): **exit section**
Framework for Analysis of Solutions

Generalized structure of every process:

```
repeat
  preceding section
  entry section
  critical section
  exit section
  following section
forever
```

Framework (cont)

- More than one CPU may be present
- Hardware "serializes" accesses to memory
- Each process operates at non-zero speed, but no other assumptions about speed
- No assumptions about the order of interleaved execution
Valid Solution: Classic Requirements

1. Mutual Exclusion: at any moment, at most one process can be in its critical section (CS)

2. Progress: if no process is executing in its CS, a process that requests entry should be allowed to enter its CS immediately

3. Bounded Waiting: upper bound on how long any process is forced to wait for entry to CS

Valid Solution: Stallings

1. Mutual Exclusion must be enforced: at any moment, at most one process can be in its CS

2. A process which halts outside its CS must not interfere with other processes

3. No indefinite delay: deadlock and starvation cannot occur
Valid Solution: Stallings (cont)

4. When no process is executing in its CS, a process that requests entry should be allowed to enter its CS immediately

5. No assumptions about number (or relative speed) of processors

6. A process remains in its CS for a finite amount of time

Types of Solutions

- Software solutions: algorithms which do not rely on any other assumptions beyond framework described earlier

- Hardware solutions: rely on special machine instructions

- OS solutions: provide system calls and data structures in the kernel for use by user programs
Software Solutions

- Peterson’s Algorithm works for two processes
- Can be generalized to more processes (Lamport’s Bakery Algorithm)

Notation:
- Two processes: P0 and P1
- When discussing process Pi, Pj always denotes the other process (i ≠ j)

Peterson’s Algorithm

```
repeat
    preceding section
    flag[i] = true;
    turn = j;
    do {} while (flag[j] and (turn==j));

    critical section
    flag[i] = false;

    following section
    forever
```
Drawbacks of Software Solutions

- Processes that are requesting to enter their CS are busy waiting (consuming CPU cycles needlessly)

- If critical sections are long, it would be more efficient to block those processes that are waiting

- Note: busy waiting is acceptable if a critical section is short (such as in the kernel)

Hardware Solutions: Disable Interrupts

- Uniprocessor: mutual exclusion is preserved but efficiency of execution is degraded since all other processes are prohibited from interrupting

- Multiprocessor: mutual exclusion is not preserved

Process Pi:

repeat
    preceding sec
disable interrupts
    critical section
    enable interrupts
    following sec
    forever
Hardware Solutions: Special Instructions

- Hardware already serializes memory access (if two accesses are attempted at the same time, the hardware does them one at a time)

- Create machine instructions that perform two actions \textit{atomically (indivisibly)} on the same memory location (ex: reading and writing)

- Enforces mutual exclusion (more complex logic to meet requirements of CS problem)

SPARC SWAP instruction

- Assembly language notation:

  \texttt{swap [address], register}

- Definition: the SWAP instruction atomically exchanges the contents of the register (4 bytes) and the contents of the word (4 bytes) at the addressed memory location. In a multiprocessor system, two or more processors addressing the same memory location simultaneously are guaranteed to access it in an undefined, but serial, order.
Mutual Exclusion Using Swap Instruction

- Shared variable Bolt initialized to 0
- The first $P_i$ which finds Bolt equal to 0 enters CS

Process $i$:

```
repeat
    preceding section
    Key = 1
    do Swap( &Key, &Bolt ) while Key != 0;
    critical section
    Bolt = 0;
    following section
forever
```

Drawbacks of Hardware Solutions

- Processes that are requesting to enter their CS are busy waiting (consuming CPU cycles needlessly)

- Does not meet all criteria for valid solution to Critical Section problem:
  - starvation is possible (choice of next process is arbitrary)
  - deadlock is possible (when OS uses priority scheduling)
Summary:

- Software solutions to Critical Section Problem using normal instructions are possible, but use busy waiting and are somewhat complex when extended to more than two processes.

- Hardware solutions using special "atomic" instructions are possible, but use busy waiting and are somewhat complex when extended to ensure bounded waiting.

Semaphores

- A semaphore is a special variable that, apart from initialization, can only be accessed via atomic and mutually exclusive operations:
  - wait(S) -- sometimes P(S)
  - signal(S) -- sometimes V(S) or post(S)

- When a process has to wait, it will be put in a queue of processes which are blocked on that same semaphore.
Semaphore operations (atomic)

Semaphore implemented as a record with two fields:

- count -- integer
- queue -- list of blocked processes

```c
struct semaphore {
    int count;
    queueType queue;
};
```

```c
void wait( semaphore s ) {
    s.count--;
    if (s.count < 0)
        // put process in s.queue, block it
};
```

```c
void signal( semaphore s ) {
    s.count++;
    if (s.count <= 0)
        // get some process from s.queue
};
```
Semaphores: Observations

- When S.count >= 0: the number of processes that can execute wait(S) without being blocked equals S.count

- When S.count < 0: the number of processes waiting on S equals |S.count|

Semaphores: Observations

- Atomicity and mutual exclusion: must ensure that only one process is executing code in wait(S) or signal(S) for a given semaphore S

- Hence, the blocks of code defining wait(S) and signal(S) are critical sections and must be guarded
Semaphores: Observations

- The critical sections defined by wait(S) and signal(S) are very short (~10 instructions)
- Use one of the software or hardware schemes discussed previously
- Amount of busy waiting should be very small (and thus acceptable)

Semaphores for Critical Sections

Initialize S to 1

```
repeat
    preceding section
    wait(S)
    critical section
    signal(S)
    following section
forever
```
Normal execution can proceed in parallel, but execution in critical sections is serialized:

Semaphores for Synchronization

- We have two processes: P1 and P2
- Statement S1 in P1 needs to be performed before statement S2 in P2
- Initialize semaphore Synch to 0

```
P1: S1
    signal(Synch)

P2: wait(Synch)
    S2
```