Lecture Topics

- Today: Concurrency: Mutual Exclusion  
  (Stallings, chapter 5.1-5.4, 5.7)
- Next: Concurrency: Deadlock and Starvation  
  (Stallings, chapter 6.1, 6.6-6.8)

Announcements

- Self-Study Exercise #5
- Project #3 (due 9/27)
- Project #4 (due 10/11)
Exam #1

- Thursday, 10/4 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!).

```cpp
void echo()
{
    cin >> A;
    cout << A;
}
```
Simple Example (continued)

- 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 (continued)

<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)

Example: Threads (continued)

- 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 (continued)

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

  \[
  \text{mov} \quad 0x0(\%\text{rip}),\%\text{eax} \\
  \text{add} \quad $0x1,\%\text{eax} \\
  \text{mov} \quad \%\text{eax},0x0(\%\text{rip})
  \]

Example: Threads (continued)

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 ); /* Entry section */
        Count++; /* Critical section */
        sem_post( &lock ); /* Exit section */
    }
    return NULL;
}
```

Examples:
Source code for both examples available on the course website:

~cse410/Examples/Threads/threads5.c
~cse410/Examples/Threads/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 (continued)

- 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 (continued)

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
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)