COMP750

Threads and Processes

Creation of New Processes

- Processes are created by fork( )
- fork( ) returns:
  - 0 if this is the new child process
  - PID of child if this is the parent process
- Microsoft uses the spawn functions to create a new process.

OS action with a fork

- Memory is allocated for the new process.
- The parent address space is copied.
- The parent Process Control Block is copied.
- A new PID is created for the child and stored in the new PCB.
- The child process is put on the scheduling queue.

Threads

- A process is a unit of resource ownership
- A thread is a unit of dispatching.
- A thread is another flow of control through the same program.
- Threads have an entry in the scheduling queue.
- Each thread has it’s own stack.

Comparison of Threads and Processes

- Cheaper to create a new thread
- Easier to switch between threads in the same process than between different processes
- Threads can easily share data
- Processes are protected from one another.

Execution Assumptions

- Reads and writes to basic data types are atomic
- Values to be manipulated are loaded into registers, changes, then stored back to memory.
- Each process has its own set of registers
- Any intermediate results of a complex expression are stored in private memory (i.e. stack)

Mutual Exclusion Needs

- Each thread has its own copy of the CPU registers (either separate CPU or switched use of a single CPU).
- Shared values exist once in RAM.
- Threads are assumed to execute at a nonzero, but unknown, rate.
- Thread switching can occur at any time.

Multiple Thread Conflicts

- A thread can be halted at any time by the OS to run another thread.
- There is no guarantee of relative thread speed. Each thread executes at an unknown speed.
- The execution of multiple threads can be interleaved in many ways.

Nondeterministic Execution

- When multiple threads access shared data, the results may be nondeterministic.
- Consider two threads executing in parallel

```
Thread 1            Thread 2
x = 13;             x = 47;
```

- **Depending upon which thread the OS executes first, the final value of x may be 13 or 47.**

Sequentially Equivalent

- We consider the results of the parallel execution of two threads to be ***Sequentially Equivalent*** if the results are equal to thread 1 running to completion and then thread 2 running or thread 2 running to completion and then thread 1.
- Results not equal to running the threads sequentially are not Sequentially Equivalent
- We often consider the results of parallel execution to be *"correct"* if it is sequentially equivalent.

Erroneous Results

- Sometimes when multiple threads access shared data, the results may not be sequentially equivalent.
- Each thread has its own set of registers values. When the OS switches between threads, it saves and restores the registers
- Values in the register may not match RAM is another thread changed a variable.

X++ in Machine Language

Thread 1	Thread 2
LOAD R1,X	
ADD R1,1	
OS switches to another thread	
	LOAD R1,X
	ADD R1,1
	STORE R1,X
OS switches to another thread	
STORE R1,X	

Mutual Exclusion

- To avoid problems when sharing data between threads, each thread has to have exclusive access to the data.
- A segment of code that can only be executed by one thread at a time is called a ***critical section***.
- If a second thread attempts to execute the critical section while another thread is already executing there, the second thread will be suspended until the first thread exits the critical section.

Mutual Exclusion Format

- The general format for creating a mutually exclusive critical section is:

…

Critical Section Entry code

Critical Section

Critical Section Entry code

…

- **Some languages do this automatically**

Mutual Exclusion Techniques

- Busy Waiting
- Disabling interrupts
- Synchronized methods or blocks
- Monitors
- Semaphores
- Test and Set instructions

Busy Waiting

- With busy waiting, a program executes in a tight loop waiting for the critical section to become available.
- The Dekker and Peterson algorithms provide mutual exclusion through busy waiting.
- Busy waiting uses lots of CPU time that would be better spent running other threads.
- Programmers should avoid busy waiting.

Synchronized Methods

- Java and C# support synchronized methods.

public synchronized int myfunc(char x);

- Only one thread at a time can execute in any synchronized method of an object.
- If another thread calls that or any other synchronized method in the object, its execution will be suspended until the first thread returns from the method.

Synchronized Method Example

```
public class COMP750 {
  private int numThings = 0;
  public synchronized void add(int x) {
     numThings += x;
  }
  public synchronized void sub(int x) {
     numThings -= x;
  }
}
```

Semaphores

- Semaphores were originally described by E. Dykstra.
- A semaphore has an integer counter and a queue of threads suspended on the semaphore.
- Semaphores can only be modified by two functions P *(or wait)* and V *(or signal)*.

P or Wait Function

```
void P(semaphore s) {
   while (s.counter == 0) {/* nothing */}
   s.counter--;
}
```

- If the semaphore counter is zero, the thread waits until it is nonzero.
- The counter is decremented.

V or Signal Function

```
void V(semaphore s) {
   s.counter++;
}
```

- Increments semaphore counter.
- If another thread was waiting in the P function, it will be allowed to continue.

OS Semaphore Implementation

- The previous definitions of the P and V functions used busy waiting.
- A better implementation asks the OS to suspend the thread.

Better P and V

```
void P(semaphore s) {
   if (s.counter == 0) {put thread on s.queue}
   s.counter--;
}
void V(semaphore s) {
   s.counter++;
   if (s.queue != NULL) {activate one thread}
}
```

Serially Equivalent

- The outcome of the parallel execution of two processes X & Y is sequentially equivalent if the results are equal to execution of X then Y or the execution of Y then X.

Atomic Actions

- The intermediate results of an atomic action are never visible.
- Atomic actions appear to either occur completely or not occur at all.
- Atomic action are never partially complete.

Atomic Actions

- Expressions involving local data are always atomic.
- Expressions meeting the at-most-once property are atomic.

At-Most-Once property

- An expression *e* satisfies the at-most-once property if it refers to at most one simple variable *y* that might be changed by another process while *e* is being evaluated and it refers to *y* at most once. An assignment *x* = *e* satisfies the at-most-once property either if *e* satisfies the property and *x* is not read by another process or if *e* does not refer to any variable that might be changed.