Concurrency and Synchronization

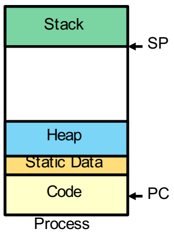
ECE 650
Systems Programming & Engineering
Duke University, Spring 2016

Concurrency

- Multiprogramming
  - Supported by most all current operating systems
  - More than one “unit of execution” at a time
- Uniprogramming
  - A characteristic of early operating systems, e.g. MS/DOS
  - Easier to design; no concurrency
- What do we mean by a “unit of execution”?

Process vs. Thread

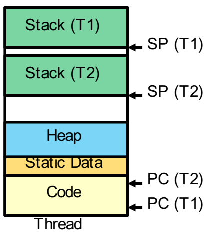
- Process vs. Thread



- A process is –
  - Execution context
    - Program counter (PC)
    - Stack pointer (SP)
    - Registers
  - Code
  - Data
  - Stack

Process vs. Thread

- Process vs. Thread



- A thread is –
  - Execution context
    - Program counter (PC)
    - Stack pointer (SP)
    - Registers

Process vs. Thread

- Process: unit of allocation
  - resources, privileges, etc.
- Thread: unit of execution
  - PC, SP, registers
- Thread is a unit of control within a process
- Every process has one or more threads
- Every thread belongs to one process

Process Execution

- When we execute a program
  - OS creates a process
  - Contains code, data
  - OS manages process until it terminates
    - We will talk more about these aspects of process management
    - E.g. scheduling, system calls, etc.
- Every process contains certain information
  - ID
  - Process state (for scheduling purposes)
  - Program counter, stack pointer, CPU registers
  - Memory management info, files, I/O

Process Execution (2)

- A process is created by the OS via system calls
  - fork(): make exact copy of this process and run
    - Forms parent/child relationship between old/new process
    - Return value of fork indicates the difference
    - Child returns 0; parent returns child's PID
  - exec(): can follow fork() to run a different program
    - Exec takes filename for program binary from disk
    - Loads new program into the current process's memory
- A process may also create & start execution of threads
  - Many ways to do this
  - System call: clone(); Library call: pthread_create()

Back to Concurrency…

- We have multiple units of execution, but single resources
  - CPU, physical memory, IO devices
  - Developers write programs as if they have exclusive access
- OS provides illusion of isolated machine access
  - Coordinates access and activity on the resources

How Does the OS Manage?

- Illusion of multiple processors
  - Multiplex threads in time on the CPU
  - Each virtual "CPU" needs a structure to hold:
    - Program Counter (PC), Stack Pointer (SP)
    - Registers (Integer, Floating point, others…?)
  - How switch from one CPU to the next?
    - Save PC, SP, and registers in current state block
    - Load PC, SP, and registers from new state block
  - What triggers switch?
    - Timer, voluntary yield, I/O, other things
- We will talk about other management later in the course
  - Memory protection, IO, process scheduling

Concurrent Program

- Two or more threads execute concurrently
  - Many ways this may occur…
  - Multiple threads time-slice on 1 CPU with 1 hardware thread
  - Multiple threads at same time on 1 CPU with N HW threads
    - Simultaneous multi-threading
  - Multiple threads at same time on M CPUs with N HW threads
    - Chip multi-processor (CMP) or Symmetric multi-processor (SMP)
- Cooperate to perform a task
- How do threads communicate?
  - Recall they share a process context
    - Code, static data, *heap*
  - Can read and write the same memory
    - variables, arrays, structures, etc.

Motivation for a Problem

- What if two threads want to add 1 to shared variable?
  - x is initialized to 0

```
x = x + 1;
```

May get compiled into:
(x is at mem location 0x8000)

```
ldw   r1,  0(0x8000)
addi  r1,  r1,  1
stw   r1,  0(0x8000)
```

- A possible interleaving:

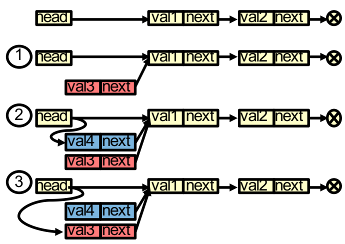
```
        P1                         P2
ldw   r1,  0(0x8000)
                          ldw   r1,  0(0x8000)
addi  r1,  r1,  1
                          addi  r1,  r1,  1
stw   r1,  0(0x8000)
                          stw   r1,  0(0x8000)
```

- At the end, x will have a value of 1 in memory!!

Another Example – Linked List

Insert at head of linked list:

```
Node new_node = new Node();
new_node->data = rand();
new_node->next = head;
head = new_node;
```



- Two concurrent threads (A & B) want to add a new element to list
  1. A executes first three instructions & stalls for some reason (e.g. cache miss)
  2. B executes all 4 instructions
  3. A eventually continues and executes 4th instruction
  - Item added by thread B is lost!

Race Conditions

- The example problems occur due to race conditions
- Race Condition
  - Result of computation by concurrent threads depends on the precise timing of the execution of an instruction sequence by one thread relative to another
- Sometimes result may be correct, sometimes incorrect
  - Depends on execution timing
  - Non-deterministic result
- Need to avoid race conditions
  - Programmer must control possible execution interleaving of threads

Mutual Exclusion

- Previous examples show problem of multiple processes or threads performing read/write ops on shared data
  - Shared data = variables, array locations, objects
- Need mutual exclusion!
  - Enforce that only one thread at a time in a code section
  - This section is also called a *critical section*
  - Critical section is set of operations we want to execute *atomically*
- Provided by lock operations

```
lock(x_lock);
x = x + 1;
unlock(x_lock);
```

- Also note => this isn't only an issue on parallel machines
  - Think about multiple threads time-sharing a single processor
  - What if a thread is interrupted after load/add but before store?

Mutual Exclusion

- Interleaving with proper use of locks (mutex)

P1

```
lock(x_lock)
ldw   r1, 0(8000)
addi  r1, r1, 1
stw   r1, 0(8000)
unlock(x_lock)
```

P2

```
lock(x_lock)
ldw   r1, 0(8000)
addi  r1, r1, 1
stw   r1, 0(8000)
unlock(x_lock)
```

- At the end, x will have a value of 2 in memory

Global Event Synchronization

- BARRIER (name, nprocs)
  - Thread will wait at barrier call until nprocs threads arrive
  - Built using lower level primitives
  - Separate phases of computation
- Example use:
  - N threads are adding elements of an array into a sum
  - Main thread is to print sum
  - Barrier prevents main thread from printing sum too early
- Use barrier synchronization only as needed
  - Heavyweight operation from performance perspective
  - Exposes load imbalance in threads leading up to a barrier

Point-to-point Event Synchronization

- A thread notifies another thread so it can proceed
  - E.g. when some event has happened
  - Typical in producer-consumer behavior
  - Concurrent programming on uniprocessors: semaphores
  - Shared memory parallel programs: semaphores or monitors or variable flags

flag

P0:

```
S1: datum = 5;
S2: datumIsReady = 1;
```

P1:

```
S3: while (!datumIsReady) {};
S4: print datum
```

monitor

P0:

```
S1: datum = 5;
S2: signal(ready);
```

P1:

```
S3: wait(ready);
S4: print datum
```


Lower Level Understanding

- How are these synchronization operations implemented?
  - Mutexes, monitors, barriers
- An attempt at mutex (lock) implementation

```
void lock (int *lockvar) {
  while (*lockvar == 1) {} ;   // wait until released
  *lockvar = 1;                // acquire lock
}

void unlock (int *lockvar) {
  *lockvar = 0;
}
```

In machine language, it looks like this:

```
lock:  ld   R1, &lockvar     // R1 = lockvar
       bnz  R1, lock         // jump to lock if R1 != 0
       st   &lockvar, #1     // lockvar = 1
       ret                   // return to caller
unlock:  st   &lockvar, #0   // lockvar = 0
         ret                 // return to caller
```

Problem

- Unfortunately, this attempted solution is incorrect
- The sequence of ld, bnz, and sti are not atomic
  - Several threads may be executing it at the same time
- It allows several threads to enter the critical section simultaneously

Time	*Thread 0*	*Thread 1*
↓	`lock: ld R1, &lockvar`	
	`bnz R1, lock`	`lock: ld R1, &lockvar`
	`sti &lockvar, #1`	`bnz R1, lock`
		`sti &lockvar, #1`

Software-only Solutions

- Peterson's Algorithm (mutual exclusion for 2 threads)

```
int turn;
int interested[n];    // initialized to 0

void lock (int process, int lvar) {    // process is 0 or 1
  int other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process && interested[other] == TRUE) {} ;
}
// Post: turn != process or interested[other] == FALSE

void unlock (int process, int lvar) {
  interested[process] = FALSE;
}
```

- Exit from lock() happens only if:
  - interested[other] == FALSE: either the other process has not competed for the lock, or it has just called unlock()
  - turn != process: the other process is competing, has set the turn to itself, and will be blocked in the while() loop

Help From Hardware

- Software-only solutions have drawbacks
  - Tricky to implement – think about more than 2 threads
  - Need to consider different solutions for different memory consistency models
- Most processors provide atomic operations
  - E.g. Test-and-set, compare-and-swap, fetch-and-increment
  - Provide atomic processing for a set of steps, such as
    - Read a location, capture its value, write a new value
  - Test-and-set
    - Instruction supported by HW
    - Write to a memory location and return its old value as a single atomic operation

Programming with Pthreads

- POSIX pthreads
  - Found on most all modern POSIX-compliant OS
  - Also Windows implementations
  - Allows a process to create, spawn, and manage threads
- How to use it:
  - Add #include <pthread.h> to your C/C++ source code
  - When compiling with gcc, add –lpthread to your list of libraries
    - gcc –o p_test p_test.c –lpthread
  - Instrument the code with pthread function calls to:
    - Create threads
    - Wait for threads to complete
    - Destroy threads
    - Synchronize across threads
    - Protect critical sections

Pthread Thread Creation

- Create a pthread:
  - int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
- Arguments:
  - pthread_t *thread_name – thread object (contains thread ID)
  - pthread_attr_t *attr – attributes to apply to this thread
  - void *(*start_routine)(void *) – pointer to function to execute
  - void *arg – arguments to pass to above function
- Example:

```
pthread_t   *thrd;
pthread_create(thrd,   NULL,  &do_work_fcn,   NULL);
```


Pthread Destruction

- pthread_join(pthread_t thread, void **value_ptr)
  - Suspends the calling thread
  - Waits for successful termination of the specified thread
  - value_ptr is optional data passed from terminating thread's exit
- pthread_exit(void *value_ptr)
  - Terminates a thread
  - Provides value_ptr to any pending pthread_join() call

Pthread Mutex

- pthread_mutex_t lock;
- Initialize a mutex; 2 ways:
  - int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutex_attr);
  - pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    - Initialized with default pthread mutex attributes
    - This is typically good enough
- Operate on the lock:
  - int pthread_mutex_lock(pthread_mutex_t* mutex);
  - int pthread_mutex_trylock(pthread_mutex_t* mutex);
  - int pthread_mutex_unlock(pthread_mutex_t* mutex);

Pthread Barrier

- pthread_barrier_t barrier;
- Initialize a barrier; 2 ways:
  - int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *barrier_attr, unsigned int count);
  - pthread_barrier_t barrier = PTHREAD_BARRIER_INITIALIZER(count);
    - Initialized with default pthread barrier attributes
    - This is typically good enough
- Operation on a barrier:
  - int pthread_barrier_wait(pthread_barrier_t *barrier);

Pthread Example (Matrix Mul)

```
double **a, **b, **c;
int numThreads, matrixSize;

int main(int argc, char *argv[]) {
    int i, j;
    int *p;
    pthread_t *threads;

    // Initialize numThreads, matrixSize; allocate and init a/b/c matrices
    // ...

    // Allocate thread handles
    threads = (pthread_t *) malloc(numThreads * sizeof(pthread_t));

    // Create threads
    for (i = 0; i < numThreads; i++) {
        p = (int *) malloc(sizeof(int));
        *p = i;
        pthread_create(&threads[i], NULL, worker, (void *)(p));
    }
    for (i = 0; i < numThreads; i++) {
        pthread_join(threads[i], NULL);
    }
    printMatrix(c);
}
```


Pthread Example (Matrix Mul) cont.

```
void mm(int myId) {
    int i,j,k;
    double sum;
    // compute bounds for this thread
    int startrow = myId * matrixSize/numThreads;
    int endrow = (myId+1) * (matrixSize/numThreads) - 1;

    // matrix mult over the strip of rows for this thread
    for (i = startrow; i <= endrow; i++) {
        for (j = 0; j < matrixSize; j++) {
            sum = 0.0;
            for (k = 0; k < matrixSize; k++) {
                sum = sum + a[i][k] * b[k][j];
            }
            c[i][j] = sum;
        }
    }
}

void *worker(void *arg){
    int id = *((int *) arg);
    mm(id);
    return NULL;
}
```