CS 206

Lecture 24 – Concurrency wrap

Thread Creation

Algol 68

co-begin
stmt_1
stmt_2
...
stmt_n
end

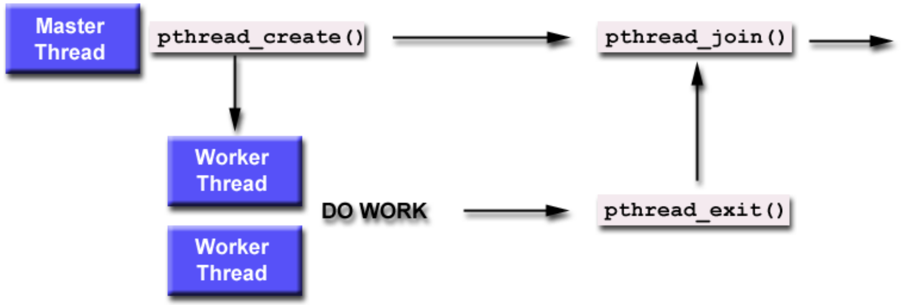
OpenMP

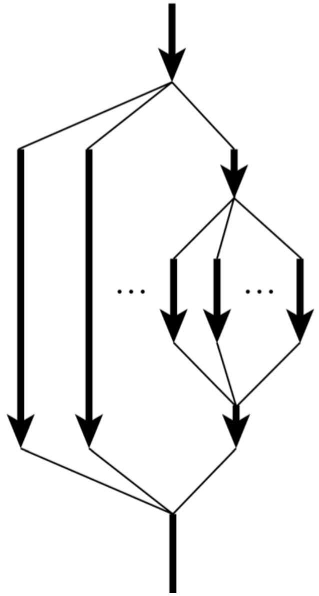
```
#pragma omp sections
{
#   pragma omp section
    { printf("thread 1 here\n"); }

#   pragma omp section
    { printf("thread 2 here\n"); }
}
```

Parallel For

```
#pragma omp parallel for
for (int i = 0; i < 3; i++) {
    printf("thread %d here\n", i);
}
```

Pthread (fork/join)





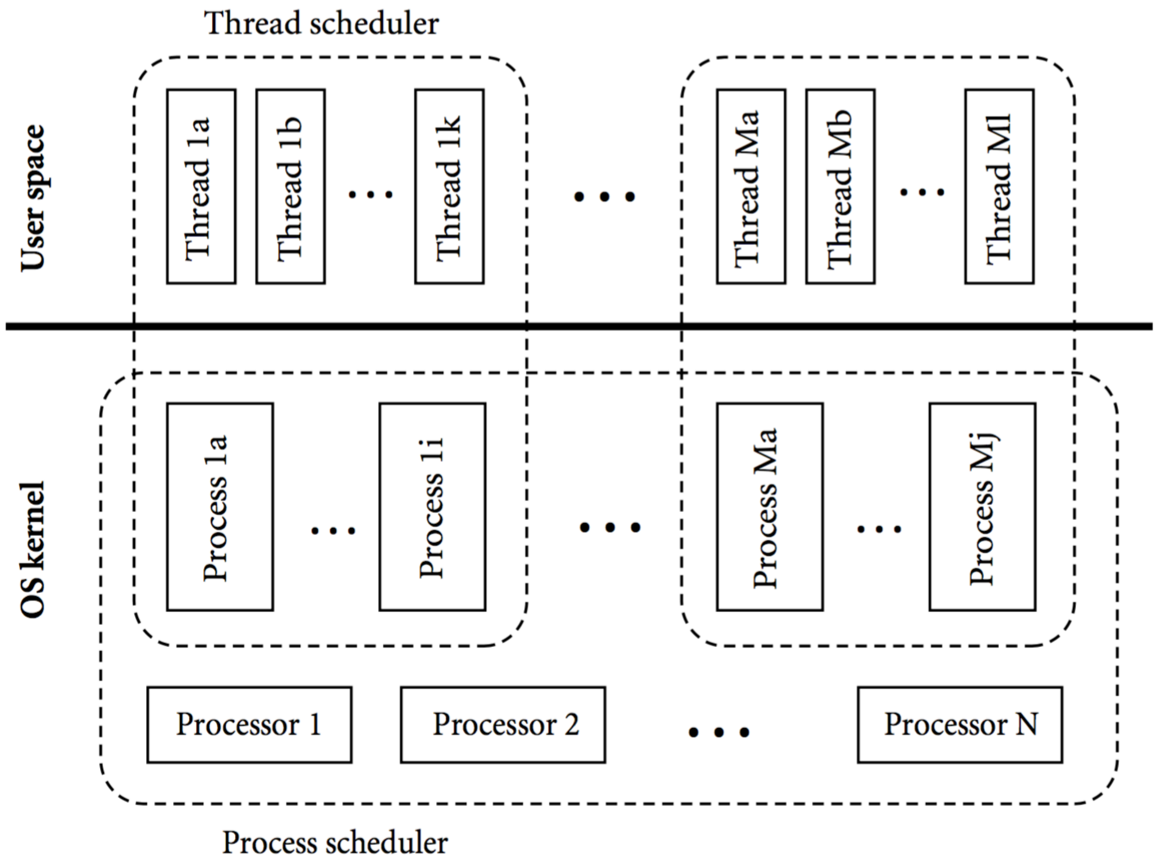
Threads in Java

- Done by constructing an object of some class derived from a predefined class – Thread
- To start executing, needs to call “start”

```
class ImageRenderer extends Thread {
    ...
    ImageRenderer( args ) {
        // constructor
    }
    public void run() {
        // code to be run by the thread
    }
}
...
ImageRenderer rend = new ImageRenderer( constructor_args );
```

Thread Implementation

- Two level implementation
  - Thread multiplexes threads on top of one or more kernel-level processes
  - Implemented as a library or language run-time package

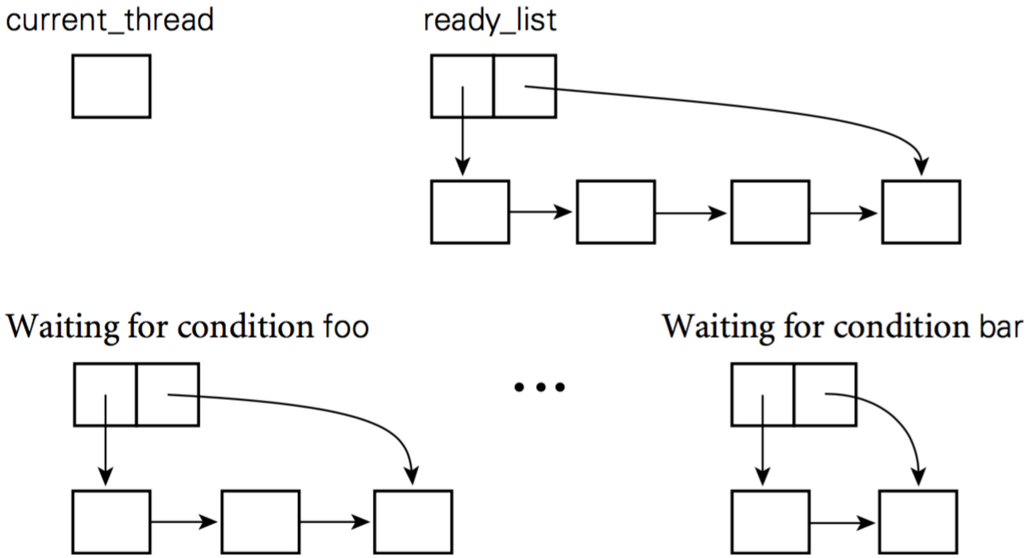


Threads Scheduling

- Can be a language feature; Cilk’s (developed at MIT) main differentiating feature was thread scheduling
- Scheduler
  - How to choose the next thread to run?
- Pre-emption mechanism
  - How to choose which thread to suspend (so others can run)
- Allow for sharing of data structures that describe a set of threads
  - So that any set of threads can run on any process
  - Data structure also called a *thread context*.

Simple Scheduling

- Scheduler maintains multiple lists (ready, blocked)
  - Threads on ready can be scheduled to run
  - Threads on blocked list are waiting for certain events to complete (locks, I/O)
    - Once event completes, they are moved to the ready list
- To yield processor to another thread, a running thread calls the scheduler
  - If the thread wants to run again in future, needs to place its context on ready list



Pre-emption

- The runtime asks OS to deliver a signal to the currently running process at a specified time in future
- OS delivers the signal by
  - saving the context (registers and pc) of the process
  - transferring control to a previously specified *handler* routine in the language run-time system
  - handler modifies state of the currently running thread; makes it appear as if the thread was about to yield
  - handler then “returns” into *yield*, which transfers control to some other thread

Language Level Synchronization

- Synchronization is principal semantic challenge for shared-memory concurrent programs.
- Need
  - Should make an operation atomic
  - Delay that operation until some precondition holds.
- Atomicity
  - Achieved by mutual exclusion locks
  - Ensures that only one thread is executing in the critical section

Locks

- Locks: variables which ensure that any such critical section executes as if it were a single atomic instruction.

```
lock_t mutex;
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock);   // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

- Coarse vs Fine grained

Implementing Locks

- Doesn't work with basic load/store primitives

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Need specialized hardware primitives to guarantee mutual exclusion

Synchronization Primitives

- The simplest hardware primitive that greatly facilitates synchronization implementations is an atomic read-modify-write

- Atomic exchange: swap contents of register and memory

- Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory

```
lock:   t&s   register, location
        bnz   register, lock
        CS
        st    location, #0
```

Busy-wait condition synchronization

- Wait and do nothing while waiting to get into the critical section
- Usually takes the form "location *X* contains value *Y* "
  - a thread needs can read *X* in a loop, waiting for *Y* to appear

- **Barriers**
  - Data-parallel algorithms are structured as a series of high-level steps/phases
  - Each thread should complete phase *i* before any can move to *i+1*
  - Typically implemented as globally shared counters, modified by an atomic *fetch_and_decrement* instructions

Semaphores

- Semaphore is an object with an integer value that we can manipulate with two routines
  - Semaphores are signaling mechanisms which can allow one or more threads/processors to access a section
- A semaphore is basically a counter with two associated operations, P and V
- A thread that calls P atomically decrements the counter and then waits until it is non-negative (*sem_wait*() in POSIX)
- A thread that calls V atomically increments the counter and wakes up a waiting thread, if any (*sem_post*() in POSIX)

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

```
sem_t m;
sem_init(&m, 0, X);

sem_wait(&m);
// critical section here
sem_post(&m);
```

Synchronization in Java

- every object accessible to more than one thread has an implicit mutual exclusion lock,
  - acquired and released by means of synchronized statements

```
synchronized (my_shared_obj) {
...     // critical section
}
```

- Synchronized statements that refer to different objects may proceed concurrently
- Within a *synchronized* statement a thread can suspend itself by calling method wait
- To resume a thread that is suspended on a given object
  - Need to call the predefined method notify from within a synchronized statement or method that refers to the same object

Optimistic Concurrency Control: Transactional Memory

- (Pessimistic) thread synchronization constructs such as locks
  - are pessimistic and prohibit threads that are outside a critical section from making any changes
  - Might have significant performance overheads
- Transactional Memory provides optimistic concurrency control by allowing threads to run in parallel with minimal interference
- A transaction is a collection of operations that can execute and commit changes as long as a conflict is not present.
  - Similar to database transactions
- Can be implemented in either hardware or software
  - Hardware: Sun’s Rock processor
  - Software: Number of libraries for C/C++, Java, C#, etc.

Transactional Memory

- Example

```
atomic {
    if (queueSize > 0) {
        remove item from queue and use it
    } else {
        retry
    }
}
```

Language Level Implementations: Monitors

- A monitor is a module or object with operations, internal state, and a number of *condition variables*
- Only one operation of a given monitor is allowed to be active at a given point in time.
- A thread that calls a busy monitor is automatically delayed until the monitor is free.
- On behalf of its calling thread, any operation may suspend itself by *wait*ing on a condition variable.
- An operation may also *signal* a condition variable, in which case one of the waiting threads is resumed, usually the one that waited first.