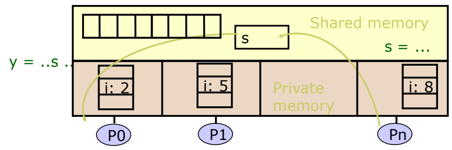
Parallel Programming with Threads

CS 240A
Tao Yang, 2013

Thread Programming with Shared Memory

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables



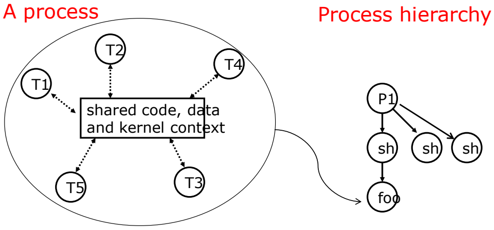
2

Logical View of Threads

- Threads associated with a process

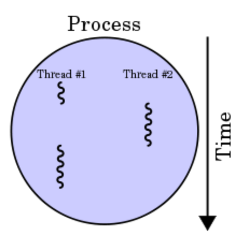
A process

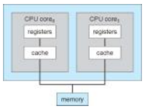
Process hierarchy



Benefits of multi-threading

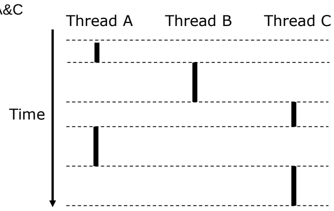
- Responsiveness
- Resource Sharing
  - Shared memory
- Economy
- Scalability
  - Explore multi-core CPUs





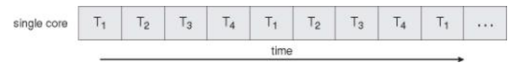
Concurrent Thread Execution

- Two threads run concurrently (are concurrent) if their logical flows overlap in time
- Otherwise, they are sequential (we'll see that processes have a similar rule)
- Examples:
  - Concurrent: A & B, A&C
  - Sequential: B & C

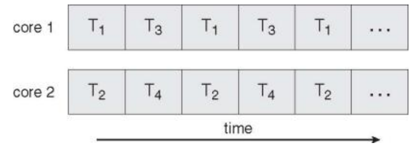


Execution Flow

Concurrent execution on a single core system

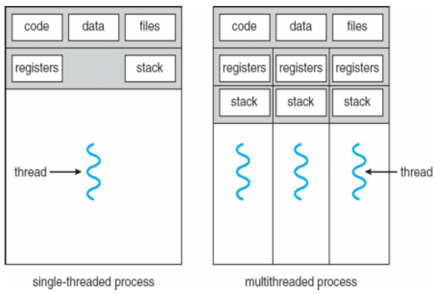


Parallel execution on a multi-core system



Difference between Single and Multithreaded Processes

Shared memory access for code/data
Separate control flow -> separate stack/registers



Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently
  - Each is context switched
- How threads and processes are different
  - Threads share code and data, processes (typically) do not
  - Threads are somewhat cheaper than processes with less overhead

Shared Memory Programming

Several Thread Libraries/systems

- PTHREADS is the POSIX Standard
  - Relatively low level
  - Portable but possibly slow; relatively heavyweight
- OpenMP standard for application level programming
  - Support for scientific programming on shared memory
  - http://www.openMP.org
- TBB: Thread Building Blocks
  - Intel
- CILK: Language of the C "ilk"
  - Lightweight threads embedded into C
- Java threads
  - Built on top of POSIX threads
  - Object within Java language

Common Notions of Thread Creation

- cobegin/coend

```
cobegin
    job1(a1);
    job2(a2);
coend
```

- Statements in block may run in parallel
- cobegins may be nested
- Scoped, so you cannot have a missing coend

- fork/join

```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

- Forked procedure runs in parallel
- Wait at join point if it's not finished

- future

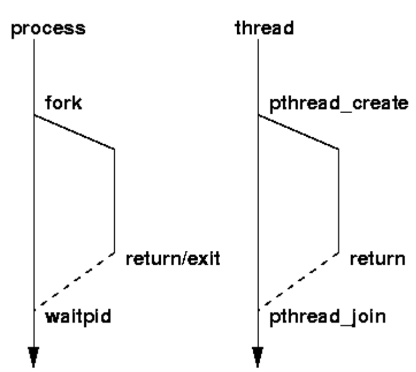
```
v = future(job1(a1));
… = …v…;
```

- Future expression evaluated in parallel
- Attempt to use return value will wait

Overview of POSIX Threads

- POSIX: ***P**ortable **O**perating **S**ystem **I**nterface for UNI**X***
  - Interface to Operating System utilities
- PThreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - In CSIL, compile a c program with gcc -lpthread
- PThreads contain support for
  - Creating parallelism and synchronization
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

Pthreads: Create threads



Forking Posix Threads

Signature:

```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (*)(void *),
                   void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute
                         &thread_fun; &fun_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attribute: minimum stack size
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

13

Some More Pthread Functions

- **`pthread_yield();`**
  - Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.
- **`pthread_exit(void *value);`**
  - Exit thread and pass value to joining thread (if exists)
- **`pthread_join(pthread_t *thread, void **result);`**
  - Wait for specified thread to finish. Place exit value into *result.

Others:

- **`pthread_t me; me = pthread_self();`**
  - Allows a pthread to obtain its own identifier pthread_t thread;

Pthreads: 14

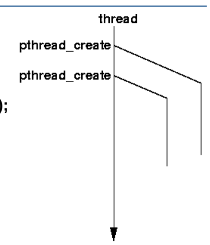
1/5/2013

Posix Threads (Pthreads) Interface

- Creating and reaping threads
  - `pthread_create, pthread_join`
- Determining your thread ID
  - `pthread_self`
- Terminating threads
  - `pthread_cancel, pthread_exit`
  - `exit` [terminates all threads] , `return` [terminates current thread]
- Synchronizing access to shared variables
  - `pthread_mutex_init, pthread_mutex_[un]lock`
  - `pthread_cond_init, pthread_cond_[timed]wait`

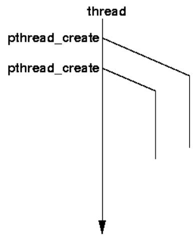
Example of Pthreads

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void * id){
  printf("Thread%d: Hello World!\n", id);
}

void main (){
  pthread_t thread0, thread1;
  pthread_create(&thread0, NULL, PrintHello, (void *) 0);
  pthread_create(&thread1, NULL, PrintHello, (void *) 1);
}
```

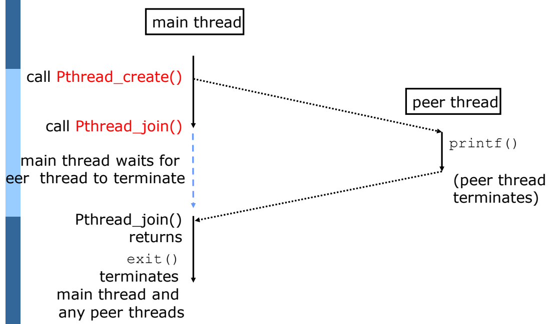


Example of Pthreads with join

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void * id){
  printf("Thread%d: Hello World!\n", id);
}

void main (){
  pthread_t thread0, thread1;
  pthread_create(&thread0, NULL, PrintHello, (void *) 0);
  pthread_create(&thread1, NULL, PrintHello, (void *) 1);
  pthread_join(thread0, NULL);
  pthread_join(thread1, NULL);
}
```



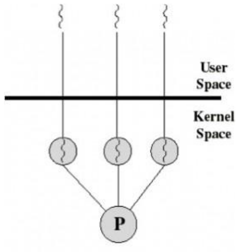
Execution of Threaded "hello, world"



Types of Threads: Kernel vs user-level

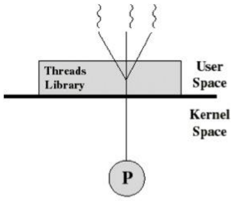
Kernel Threads

- Recognized and supported by the OS Kernel
- OS explicitly performs scheduling and context switching of kernel threads



User-level Threads

- Thread management done by user-level threads library
  - OS kernel does not know/recognize there are multiple threads running in a user program.
  - The user program (library) is responsible for scheduling and context switching of its threads.
- Examples:
  - Java threads



Recall Data Race Example

```
static int s = 0;

Thread 1                    Thread 2

  for i = 0, n/2-1            for i = n/2, n-1
      s = s + f(A[i])             s = s + f(A[i])
```

- Also called critical section problem.
- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Synchronization Solutions

1. Locks (mutex)

```
acquire lock
        critical section
release lock
        remainder section
```

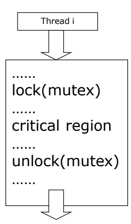
2. Semaphore

3. Conditional Variables

4. Barriers

Synchronization primitive: Mutex

```
pthread_mutex_t mutex;
const pthread_mutexattr_t attr;
int status;

status =
pthread_mutex_init(&mutex,&attr);

status =
pthread_mutex_destroy(&mutex);

status = pthread_mutex_unlock(&mutex);

status = pthread_mutex_lock(&mutex);
```



Semaphore: Generalization from locks

- Semaphore *S* – integer variable
- Can only be accessed /modified via two indivisible (atomic) operations
  - wait (S) { //also called P()

```
    wait (S) {      //also called P()
        while S <= 0
            ; // wait
        S--;
    }
    post(S) {     //also called V()
        S++;
    }
```

Semaphore for Pthreads

int status,pshared;

sem_t sem;

unsigned int initial_value;

status = sem_init(&sem,pshared,initial_value);

status = sem_destroy(&sem);

status = sem_post(&sem);

-increments (unlocks) the semaphore pointed to by *sem*

status = sem_wait(&sem);

-decrements (locks) the semaphore pointed to by *sem*

Deadlock and Starvation

- **Deadlock** – two or more processes (or threads) are waiting indefinitely for an event that can be only caused by one of these waiting processes
- **Starvation** – indefinite blocking. A process is in a waiting queue forever.
  - Let S and Q be two locks:

P_0	P_1
Acquire(S);	Acquire(Q);
Acquire (Q);	Acquire (S);
.	.
.	.
.	.
Release (Q);	Release(S);
Release (S);	Release(Q);

Deadlock Avoidance

- **Order the locks and always acquire the locks in that order.**
- **Eliminate circular waiting**

P_0	P_1
Acquire(S);	Acquire(S);
Acquire(Q);	Acquire (Q);
.	.
.	.
.	.
Release(Q);	Release (Q);
Release(S);	Release (S);

Synchronization Example for Readers-Writers Problem

- A data set is shared among a number of concurrent processes.
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Requirement:
  - allow multiple readers to read at the same time.
  - Only one writer can access the shared data at the same time.
- Reader/writer access permission table:

	Reader	Writer
Reader	OK	No
Writer	NO	No

Readers-Writers (First try with 1 lock)

- writer

```
do {
    wrt.Acquire(); //  wrt  is a lock
    //    writing is performed
    wrt.Release();
} while (TRUE);
```

- Reader

```
do {
    wrt.Acquire(); // Use wrt lock
    //    reading is performed
    wrt.Release();
} while (TRUE);
```

	Reader	Writer
Reader	?	?
Writer	?	?

Readers-Writers (First try with 1 lock)

- writer

```
do {
    wrt.Acquire(); //  wrt  is a lock
    //    writing is performed
    wrt.Release();
} while (TRUE);
```

- Reader

```
do {
    wrt.Acquire(); // Use wrt lock
    //    reading is performed
    wrt.Release();
} while (TRUE);
```

	Reader	Writer
Reader	NO	NO
Writer	NO	NO

2nd try using a lock + readcount

- writer

```
do {
    wrt.Acquire(); // Use wrt lock
    //    writing is performed
    wrt.Release();
} while (TRUE);
```

- Reader

```
do {
    readcount++; // add a reader counter.
    if(readcount==1) wrt.Acquire();
    //    reading is performed
    readcount--;
      if(readcount==0)  wrt.Release();
} while (TRUE);
```


You may also use a binary semaphore

- writer

```
do {
    wrt.P(); // Use wrt semaphore  with initial value=1
    //    writing is performed
    wrt.V();
} while (TRUE);
```

- Reader

```
do {
    readcount++; //initial value=0
    if(readcount==1) wrt.P();
    //    reading is performed
    readcount--;
      if(readcount==0)  wrt.V();
} while (TRUE);
```

What’s wrong with this?

readcount is not protected

Readers-Writers Problem with semaphone

- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers Problem (textbook)

- The structure of a writer process

```
do {
        wrt.P() ;  //Lock wrt

        // writing is performed

        wrt.V() ;  //Unlock wrt
} while (TRUE);
```


Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        mutex.P() ;
        readcount ++ ;
        if (readcount == 1)
              wrt.P() ;
        mutex.V()
           // reading is performed

        mutex.P() ;
        readcount  - - ;
        if (readcount  == 0)
              wrt.V() ;
        mutex.V() ;
} while (TRUE);
```


Synchronization Primitive: Condition Variables

- Used together with a lock
- One can specify more general waiting condition compared to semaphores.
- Avoid busy waiting in spin locks

  Let the waiting thread be blocked, placed in a waiting queue, yielding CPU resource to somebody else.

Pthread synchronization: Condition variables

int status;   pthread_condition_t cond;

const pthread_condattr_t attr;

pthread_mutex mutex;

status = pthread_cond_init(&cond,&attr);

status = pthread_cond_destroy(&cond);

status = pthread_cond_wait(&cond,&mutex);

-wait in a queue until somebody wakes up. Then the mutex is reacquired.

status = pthread_cond_signal(&cond);

- wake up one waiting thread.

status = pthread_cond_broadcast(&cond);

- wake up all waiting threads in that condition

How to Use Condition Variables: Typical Flow

- Thread 1

Lock(mutex);

While (condition is not satisfied)

Wait(mutex, cond);

Critical Section;

Unlock(mutex)

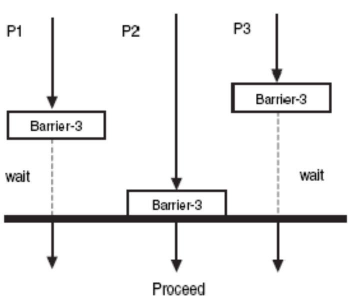
- Thread 2:

Lock(mutex);

When condition can satisfy, Signal(mylock);

Unlock(mutex);

Synchronization primitive: Barriers



Barrier in Pthreads

Barrier -- global synchronization

- Especially common when running multiple copies of the same function in parallel
  - SPMD "Single Program Multiple Data"
- simple use of barriers -- all threads hit the same one

```
work_on_my_subgrid();
barrier;
read_neighboring_values();
barrier;
```

- more complicated -- barriers on branches (or loops)

```
if (tid % 2 == 0) {
  work1();
  barrier
} else { barrier }
```

- barriers are not provided in all thread libraries

Creating and Initializing a Barrier

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```
pthread_barrier_t b;
pthread_barrier_init(&b,NULL,3);
```

- The second argument specifies an attribute object for finer control; using NULL yields the default attributes.
- To wait at a barrier, a process executes:

```
pthread_barrier_wait(&b);
```

41

Implement a simple barrier

```
int count=0;

barrier(N) { //for N threads

  count ++;

  while (count <N);
}
```

What's wrong with this?

42

What to check for synchronization

- Access to EVERY share variable is synchronized with a lock
- No busy waiting:
  - Wait when the condition is not met
  - Call condition-wait() after holding a lock/detecting the condition

Implement a barrier

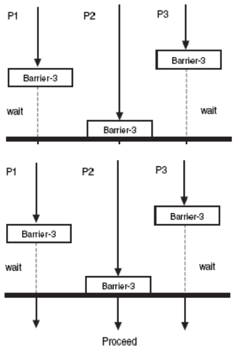
```
int count=0;

barrier(N) { //for N threads
  Lock(m);
  count ++;
  while (count <N)
           Wait(m, mycondition);
  if(count==N) {
     Broadcast(mycondition);
     count=0;
   }
  Unlock(m);
}
```

What's wrong with this?

Count=N for next barrier() called in another thread

44

Barriers called multiple times



barrier(3);

barrier(3);

Summary of Programming with Threads

- POSIX Threads are based on OS features
  - Can be used from multiple languages (need appropriate header)
  - Familiar language for most of program
  - Ability to shared data is convenient
- Pitfalls
  - Data race bugs are very nasty to find because they can be intermittent
  - Deadlocks are usually easier, but can also be intermittent
- OpenMP is commonly used today as an alternative