

POSIX Threads

HUJI
Spring 2007

1

POSIX Threads

In the UNIX environment a thread:

- Exists within a process and uses the process resources.
- Has its own independent flow of control as long as its parent process exists or the OS supports it.
- May share the process resources with other threads that act equally *independently (and dependently)*.
- Dies if the parent process dies (user thread).

2

Pthread Attributes

A thread can possess an independent flow of control and be schedulable because it maintains its own:

- Stack.
- Registers. (CPU STATE!)
- *Scheduling properties (such as policy or priority)*.
- *Set of pending and blocked signals*.
- *Thread specific data*.

3

Why Threads

- Managing threads requires fewer system resources than managing processes.
 - fork() Versus pthread_create()
- All threads within a process share the same *address space*.
 - Inter-thread communication is more efficient and easier to use than inter-process communication (IPC).
- Overlapping CPU work with I/O.
- Priority/real-time scheduling.
- Asynchronous event handling.

4

Pthread Library

The subroutines which comprise the Pthreads API can be informally grouped into 3 major classes:

- **Thread management:** The first class of functions work directly on threads.
- **Mutexes:** The second class of functions deals with synchronization, called a "mutual exclusion".
- **Condition variables:** The third class of functions addresses communications between threads that share a mutex.

How to Compile?

```
#include <pthread.h>
gcc ex3.c -o ex3 -lpthread
```

5

Creating Threads

- Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

```
int pthread_create (
    pthread_t *thread,
    const pthread_attr_t *attr=NULL,
    void *(*start_routine) (void *),
    void *arg) ;
```

6

Terminating Thread Execution

- The thread returns from its starting routine (the main routine for the initial thread).
- The thread makes a call to the `pthread_exit(status)` subroutine.
- The entire process is terminated due to a call to either the `exec` or `exit` subroutines.

7

```
#define NUM_THREADS 5

void *PrintHello(void *index) {
    printf("\n%d: Hello World!\n", index);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int res, t;
    for(t=0;t<NUM_THREADS;t++){
        printf("Creating thread %d\n", t);
        res = pthread_create(&threads[t], NULL,
            PrintHello, (void *)t);
        if (res){
            printf("ERROR\n");
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

8

Joining Threads

```
int pthread_join(pthread_t thread,
    void **value_ptr);
```

- The `pthread_join()` subroutine blocks the calling thread until the specified *thread* thread terminates.
- The programmer is able to obtain the target thread's termination return status if specified through `pthread_exit(void *status)`.

9

Example Cont.

```
// main thread waits for the other threads
for(t=0;t<NUM_THREADS;t++) {
    res = pthread_join(threads[t], (void **)&status);
    if (res) {
        printf("ERROR \n");
        exit(-1);
    }
    printf("Completed join with thread %d status=
%d\n",t, status);
}
pthread_exit(NULL);
}
```

10

Few Examples

11

Example 1 - pthread_join

```
void *printme(void *ip) {
    int *i;
    i = (int *) ip;
    printf("Hi. I'm thread %d\n", *i);
    return NULL;
}

main() {
    int i, vals[4];
    pthread_t tids[4];
    void *retval;
    for (i = 0; i < 4; i++) {
        vals[i] = i;
        pthread_create(tids+i, NULL, printme, vals+i);
    }
    for (i = 0; i < 4; i++) {
        printf("Trying to join with tid %d\n", i);
        pthread_join(tids[i], &retval);
        printf("Joined with tid %d\n", i);
    }
}
```

12

Output

```
Trying to join with tid 0
Hi. I'm thread 0
Hi. I'm thread 1
Hi. I'm thread 2
Hi. I'm thread 3
Joined with tid 0
Trying to join with tid 1
Joined with tid 1
Trying to join with tid 2
Joined with tid 2
Trying to join with tid 3
Joined with tid 3
```

13

Example 2 - pthread_exit

```
void *printme(void *ip) {
    int *i;
    i = (int *) ip;
    printf("Hi. I'm thread %d\n", *i);
    pthread_exit(NULL);
}

main() {
    int i, vals[4];
    pthread_t tids[4];
    void *retval;
    for (i = 0; i < 4; i++) {
        vals[i] = i;
        pthread_create(&tids[i], NULL, printme, &vals[i]);
    }
    pthread_exit(NULL);
    for (i = 0; i < 4; i++) {
        printf("Trying to join with tid %d\n", i);
        pthread_join(tids[i], &retval);
        printf("Joined with tid %d\n", i);
    }
}
```

14

The Output

```
Hi. I'm thread 0
Hi. I'm thread 1
Hi. I'm thread 2
Hi. I'm thread 3
```

15

Example 3 - Exit

```
void *printme(void *ip) {
    int *i;
    i = (int *) ip;
    printf("Hi. I'm thread %d\n", *i);
    exit(0);
}

main() {
    int i, vals[4];
    pthread_t tids[4];
    void *retval;
    for (i = 0; i < 4; i++) {
        vals[i] = i;
        pthread_create(&tids[i], NULL, printme, &vals[i]);
    }
    for (i = 0; i < 4; i++) {
        printf("Trying to join with tid %d\n", i);
        pthread_join(tids[i], &retval);
        printf("Joined with tid %d\n", i);
    }
}
```

16

Output?

```
Trying to join with tid 0
Hi. I'm thread 0
```

17

Pthread Synchronization

18

Critical Section

Balance	1 st thread	2 nd thread
1000\$	Read balance: 1000\$	
1000\$		Read balance: 1000\$
1000\$		Deposit: 200\$
1000\$	Deposit: 200\$	
1200\$	Update balance	
1200\$		Update balance <small>19</small>

Mutex

- Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- A mutex variable acts like a "lock" protecting access to a shared data resource.
- The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time.

20

Work Flow

A typical sequence in the use of a mutex is as follows:

- Create and initialize a mutex variable.
- Several threads attempt to lock the mutex.
- Only one succeeds and that thread owns the mutex.
- The owner thread performs some set of actions.
- The owner unlocks the mutex.
- Another thread acquires the mutex and repeats the process.
- Finally the mutex is destroyed.

21

Creating and Destroying Mutex

Mutex variables must be declared with type `pthread_mutex_t`, and must be initialized before they can be used:

- Statically, when it is declared. For example:
`pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`
- Dynamically,
`pthread_mutex_init(&mutex, attr)`
This method permits setting mutex object attributes (for default setting use NULL).

`pthread_mutex_destroy(&mutex)` should be used to free a mutex object when is no longer needed.

22

Locking Mutex

- The `pthread_mutex_lock(&mutex)` routine is used by a thread to acquire a lock on the specified `mutex` variable.
- If the `mutex` is already locked by another thread, this call will block the calling thread until the `mutex` is unlocked.

23

Unlock Mutex

- `pthread_mutex_unlock(&mutex)` will unlock a `mutex` if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data.
- An error will be returned if:
 - If the `mutex` was already unlocked.
 - If the `mutex` is owned by another thread.

24

Example

```
Thread 1:          Thread 2:
a = counter;
a++;
counter = a;

b = counter;
b--;
counter = b;
```

25

Fixed Example

```
pthread_mutex_lock (&mut);
a = counter;
a++;
counter = a;
pthread_mutex_unlock (&mut);

pthread_mutex_lock (&mut);
b = counter;
b--;
counter = b;
pthread_mutex_unlock (&mut);
```

26

Conditional Variables

- While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, The programmer would need to have threads continually polling (usually in a critical section), to check if the condition is met.
- A condition variable is a way to achieve the same goal without polling (a.k.a. busy wait)

27

Condition Variables

- Useful when a thread needs to wait for a certain condition to be true.
- In **threads**, there are four relevant procedures involving condition variables:
 - `pthread_cond_init(pthread_cond_t *cv, NULL);`
 - `pthread_cond_destroy(pthread_cond_t *cv);`
 - `pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *lock);`
 - `pthread_cond_signal(pthread_cond_t *cv);`

28

Creating and Destroying Conditional Variables

- Condition variables must be declared with type `pthread_cond_t`, and must be initialized before they can be used.
- Statically, when it is declared. For example:
`pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;`
- Dynamically
`pthread_cond_init(cond, attr);`
Upon successful initialization, the state of the condition variable becomes initialized.
- `pthread_cond_destroy(cond)` should be used to free a condition variable that is no longer needed.

29

pthread_cond_wait

- `pthread_cond_wait(cv, lock)` is called by a thread when it wants to block and wait for a condition to be true.
- It is assumed that the thread has locked the mutex indicated by the second parameter.
- The thread releases the mutex, and blocks until awakened by a `pthread_cond_signal()` call from another thread.
- When it is awakened, it waits until it can acquire the mutex, and once acquired, it returns from the `pthread_cond_wait()` call.

30

pthread_cond_signal

- `pthread_cond_signal()` checks to see if there are any threads waiting on the specified condition variable. If not, then it simply returns.
- If there are threads waiting, then one is awakened.
- There can be no assumption about the order in which threads are awakened by `pthread_cond_signal()` calls.
- It is natural to assume that they will be awakened in the order in which they waited, but that may not be the case...
- Use `loop` or `pthread_cond_broadcast()` to awake all waiting threads.

31

```
typedef struct {
    pthread_mutex_t *lock;
    pthread_cond_t *cv;
    int *ndone;
    int id;
} Tstruct;
#define NTHREADS 5
void *barrier(void *arg) {
    Tstruct *ts;
    int i;
    ts = (Tstruct *) arg;
    printf("Thread %d -- waiting for barrier\n", ts->id);
    pthread_mutex_lock(ts->lock);
    *ts->ndone = *ts->ndone + 1;
    if (*ts->ndone < NTHREADS) {
        pthread_cond_wait(ts->cv, ts->lock);
    }
    else {
        for (i = 1; i < NTHREADS; i++)
            pthread_cond_signal(ts->cv);
    }
    pthread_mutex_unlock(ts->lock);
    printf("Thread %d -- after barrier\n", ts->id);
}
```

32

```
main() {
    Tstruct ts[NTHREADS];
    pthread_t tids[NTHREADS];
    int i, ndone;
    pthread_mutex_t lock;
    pthread_cond_t cv;
    void *retval;
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cv, NULL);
    ndone = 0;
    for (i = 0; i < NTHREADS; i++) {
        ts[i].lock = &lock;
        ts[i].cv = &cv;
        ts[i].ndone = &ndone;
        ts[i].id = i;
    }
    for (i = 0; i < NTHREADS; i++)
        pthread_create(&tids[i], NULL, barrier, ts+i);
    for (i = 0; i < NTHREADS; i++)
        pthread_join(tids[i], &retval);
    printf("done\n"); }
}
```

33

Output

```
Thread 0 -- waiting for barrier
Thread 1 -- waiting for barrier
Thread 2 -- waiting for barrier
Thread 3 -- waiting for barrier
Thread 4 -- waiting for barrier
Thread 4 -- after barrier
Thread 0 -- after barrier
Thread 1 -- after barrier
Thread 2 -- after barrier
Thread 3 -- after barrier
done
```

34

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

35

Bounded-Buffer Problem

- One buffers that can hold N items
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

36

Bounded-Buffer Problem – Cont.

- The structure of the **producer** process

```
while (true) {
    // produce an item

    P (empty);
    P (mutex);

    // add the item to the buffer

    V (mutex);
    V (full);
}
```

37

Bounded-Buffer Problem – Cont.

- The structure of the **consumer** process

```
while (true) {
    P (full);
    P (mutex);

    // remove an item from buffer

    V (mutex);
    V (empty);

    // consume the removed item
}
```

38

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.
- Problem
 - allow multiple readers to read at the same time.
 - Only one single writer can access the shared data at the same time.
 - If a writer is writing to the file, no reader may read it
- Shared Data
 - Data set
 - Semaphore mutex initialized to 1.
 - Semaphore wrt initialized to 1.
 - Integer readcount initialized to 0.

39

Readers-Writers Problem – Cont.

- The structure of a **writer** process

```
while (true) {
    P (wrt) ;

    // writing is performed

    V (wrt) ;
}
```

40

Readers-Writers Problem – Cont.

- The structure of a **reader** process

```
while (true) {
    P (mutex) ;
    readcount ++ ;
    if (readcount == 1) P (wrt) ;
    V (mutex)

    // reading is performed

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

Any problem??

What if a writer is waiting to write but there are readers that read all the time?

Writers are subject to starvation!

41

Writer-Priority: The Writer

- Extra semaphores and variables:
 - Semaphore read initialized to 1 – inhibits readers when a writer wants to write.
 - Integer writecount initialized to 0 – controls the setting of semaphore read.
 - Semaphore wmutex initialized to 1 – controls the updating of writecount.

```
while (true) {
    P(wmutex)
    writecount++;
    if (writecount == 1) P(read)
    V(wmutex)
    P (wrt) ;
    // writing is performed
    V (wrt) ;
    P(wmutex)
    writecount--;
    if (writecount == 0) V(read)
    V(wmutex)
}
```

42

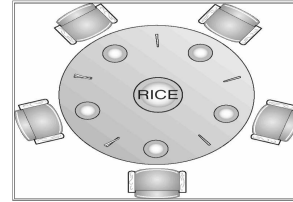
Writer-Priority: The Reader

```
while (true) {
  P(queue)
  P(read)
  P (rmutex) ;
  readcount ++ ;
  if (readcount == 1) P (wrt) ;
  V (rmutex)
  V(read)
  V(queue)
  // reading is performed
  P (rmutex) ;
  readcount -- ;
  if (readcount == 0) V (wrt) ;
  V (rmutex) ;
}
```

Queue semaphore, initialized to 1:
Since 'read' can not be a queue
(otherwise, writer won't skip
multiple readers), we have to
maintain another semaphore

43

Dining-Philosophers Problem



Shared data
Bowl of rice (data set)
Semaphore chopstick [5] initialized to 1

44

Dining-Philosophers Problem – Cont.

- The structure of Philosopher i :

```
While (true) {
  P ( chopstick[i] );
  P ( chopstick[ (i + 1) % 5] );
  // eat
  V ( chopstick[i] );
  V ( chopstick[ (i + 1) % 5] );
  // think
}
```

- Does it solve the problem?

45

Dining Philosophers Problem

- An abstract problem demonstrating some fundamental limitations of the deadlock-free synchronization
- There is no symmetric solution
- Solutions
 - Execute different code for odd/even
 - Give them another fork
 - Allow at most 4 philosophers at the table
 - Randomized (Lehmann-Rabin)

46