Recap

Multi-thread Process

- User address space
- TCB/PCB
- Concurrency -> Race condition

POSIX Thread library

- pthread_create
- pthread_exit

pthread_exit()

```
#include <pthread.h>
void pthread_exit(void *retval)
```

Terminate calling thread

- The function returns a value via retval that is available to another thread in the same process that calls pthread_join()
- The function does not return to the caller

pthread_join()

```
#include <pthread.h>
int pthread_join(pthread_t th, void **retval)
```

Wait for a thread to terminate

- th: the thread to wait for
- If retval is not NULL, then **pthread_join**() copies the exit status of the target thread into the location pointed to by retval
- Returns 0 on success, returns an error number on error
- When a thread terminates, its TCB is not deallocated until another thread performs **pthread_join()** on it

Pthread_join() Example

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

int n;

void * thread_start(void *arg)
{
  int *id = (int *)arg;
  while (n != *id);
  printf("Thread %d \n", *id);
  n--;
  pthread_exit(NULL);
}
```

```
int main()
{
  n = 0;
  int id1=1;
  int id2=2;
  int id3=3;

  pthread_t t1, t2, t3;
  printf("Parent creating threads\n");
  pthread_create(&t1, NULL, thread_start, &id1);
  pthread_create(&t2, NULL, thread_start, &id2);
  pthread_create(&t3, NULL, thread_start, &id3);
  printf("Threads created\n");
  n = 3;
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  printf("Threads are done\n");
  return 0;
}
```

Locks

Locks provide mutual exclusion to a critical section of code

- Mutual exclusion – only one thread at a time
- Critical section – a section of code that can only be executed by one thread at a time and the thread must execute the code to completion before another thread can enter

Shared variables can be accessed in a critical section

Initializing a Lock

```
int pthread_mutex_init(
          pthread_mutex_t     * mutex,
    const pthread_mutexattr_t * mutexattr)
```

This function initializes a mutex lock

- First parameter is a pointer to the mutex
- Second parameter specifies the attributes of the mutex
  - If mutexattr is NULL, default attributes are used
- Return 0 on success; otherwise, an error number is returned

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

This function destroys a mutex

- The mutex must be unlocked when called
- Attempting to destroy a locked mutex results in undefined behavior
- Return 0 on success; otherwise, an error number is returned

Using Lock to Create Critical Section

`int pthread_mutex_lock(pthread_mutex_t*mutex)`

Acquire the mutex lock

- If the mutex is unlocked, it becomes locked and owned by the calling thread
- If the mutex is already locked, the calling thread blocks until the mutex is unlocked
- Return 0 on success; otherwise, an error number is returned

`int pthread_mutex_unlock(pthread_mutex_t*mutex)`

- Release the mutex lock
- This function unlocks a mutex if called by the owning thread
  - An error will be returned if the mutex is owned by another thread
- Return 0 on success; otherwise, an error number is returned

Example

```
#include <stdio.h>
#include <pthread.h>

static volatile int counter = 0;
pthread_mutex_t mutex;


void *mythread(void *arg) {
    printf("thread %s: begin\n", (char *) arg);
    for (int i = 0; i < 1e7; i++) {

        pthread_mutex_lock(&mutex);
        counter++;

        pthread_mutex_unlock(&mutex);
    }
    printf("thread %s: end\n", (char *) arg);
    return NULL;
}


int main() {
    pthread_t p1, p2;

    printf("main: begin\n");

    pthread_mutex_init(&mutex,NULL);

    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    Pthread_mutex_destroy(&mutex);

    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Condition Variables

Condition variables are used to put a thread to sleep until another thread signals it

```
int pthread_cond_wait(
         pthread_cond_t   * cond,
         pthread_mutext_t * mutex)
```

Wait for signal

```
int pthread_cond_signal(
         pthread_cond_t   * cond)
```

Send signal

Locks

(based on Ch. 28)

Slides revised from
Matthew Tancreti
Iowa State University

Goals

Mutual Exclusion – prevent multiple threads from entering a critical section

Fairness – does each thread contending for lock get fair opportunity to enter, do not want to **starve** a thread by always giving priority to others

Performance – time overhead of entering and exiting critical section

Simple Hardware Solution – Disable Interrupts

Simple solution is to disable interrupts

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Many negative aspects

- Enable and disable interrupts are privileged instructions
- OS loses control – user program can keep CPU for as long as it wants
- Can result in important interrupts getting delayed or lost

Simple Software Software Solution?

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Simple Software Solution Has Race Condition Bug

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!)	

Peterson's Algorithm – A Software Solution That Works!

```
int flag[2];
int turn;

void init() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}
void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while it's not your turn
}
void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

Case 1: sequential execution
For example, Proc0.lock();Proc1.lock().
Proc0 finds flag[1]==0, so it locks.
Proc1 can lock only after Proc0 unlocks.

Case 2: concurrent execution

time	P0	P1
t	Flag[0]=1	Flag[1]=0
t+1	turn=1	
t+2	(block)	turn=0
t+3	lock!	(block)
t+4	unlock	(block)
t+5		lock!

Can be extended to more than 2 processes!

Disadvantage of Peterson’s Algorithm

Peterson’s solution does not work on modern computer architectures

To improve performance, processors can reorder instructions that have no dependencies

What happens if assignments to flag and turn are reordered?

Hardware Support – Test-and-Set

Common hardware support is a test-and-set instruction

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Operation of test-and-set is shown in code above, but the important point is test-and-set is not software, it is an **atomic instruction** (cannot be interrupted) so therefore no race condition possible

How to Use Test-and-Set to Build a Lock?

```
typedef struct __lock_t {
     int flag;
} lock_t;

void init(lock_t *lock) {
     // 0: lock is available, 1: lock is held
     lock->flag = 0;
}

void lock(lock_t *lock) {
     while (TestAndSet(&lock->flag, 1) == 1)
         ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
     lock->flag = 0;
}
```

Problem: Performance of Spinning

All the approaches we have seen are **spin locks**, a waiting thread keeps checking unit the lock is available

Uses CPU for indefinite amount of time

- On single CPU machine, wait for time-slice to expire
- N threads contenting for lock wait N-1 time slices

Solution to Spinning - Yield

Solution is that waiting thread should voluntarily give up CPU

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Problem: Fairness

So far, when multiple threads contending for lock the winner is up to chance

- Which ever one executes TestAndSet first

A more controlled mechanism is a FIFO queue for thread waiting on lock

Solution to Fairness - Queue

park() is system call to put thread to sleep util unpark(tid) is called

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```