Systems Programming Final

Mutex Locks

Mutexes are a primary means of implementing thread synchronization and protecting shared data when multiple writes occur.

Only one thread can own a lock at a time. No other thread can have access to data until current thread unlocks mutex. Forces threads to take turns writing and reading data. Used to prevent race conditions.

When several threads compete for a mutex, the losers block on that call. A non-blocking call is “trylock” instead of “lock”

Flag, Owner, Queue (holds suspended threads)

Basic POSIX mutex operations are:

- Initialize -- initialize a pthread_mutex_t. Second arg can be zero, meaning assume default attributes.
- lock -- If mutex is unlocked, lock it and make the calling thread the owner. If mutex is locked, then block calling thread until the mutex is unlocked. Does not return until called has the lock.
- unlock --If mutex is not locked, no effect. If locked, but caller is not owner then return EPERM. else unlock the mutex.
- trylock -- If mutex is unlocked, lock it and make the calling thread the owner. If mutex is locked, return EBUSY immediately without blocking or acquiring the lock.
- destroy -- Destroy the mutex. Don't use it after it's destroyed. Good idea to ensure mutex is unlocked before destruction.

Condition Variables

Routines that communicate between threads that share a resource. They are another way to synchronize threads. Threads synchronize based on the value of data. Always used in conjunction with Mutex.

pthread_cond_wait() -- blocks the calling thread until the specified condition is met. Called while the mutex is locked and releases mutex while it waits. After signal is received mutex is automatically locked. Programmer then must unlock it. The logical condition should always

be checked on return, as a return might not have been caused by a change in the condition.

pthread_cond_signal() -- used to signal another thread which is waiting on the condition variable. No effect if no thread is waiting
**pthread_cond_broadcast(**) -- used when more than one thread is waiting

Signal Handling

Report the occurrence of an exceptional event.

Sources: Errors, external events, and explicit requests.
Generated
Signal generated synchronously or asynchronously
Signal is generated in response to come event -- sigint, hardware exceptions, kill command, kill system call
Pending -- After generated event but not yet received by the process
Blocked -- Signals can be blocked by a process -- process doesn't receive signal -- signal left in pending state
Delivered -- blocked what the process chooses to do once signal is received
Disposition -- blocked What the process chooses to do once the signal is received:

1. Ignore signal -- can't ignore a sigkill
2. Allow a default action
3. Catch the signal -- tell OS to call a function when signal occurs.

Signal Mask -- aka Bit field -- the set of blocked signals, those that we will not receive.

Examples:

- A program error like dividing by 0
- a termination of a child process
- A call to kill by same process

Multithreading

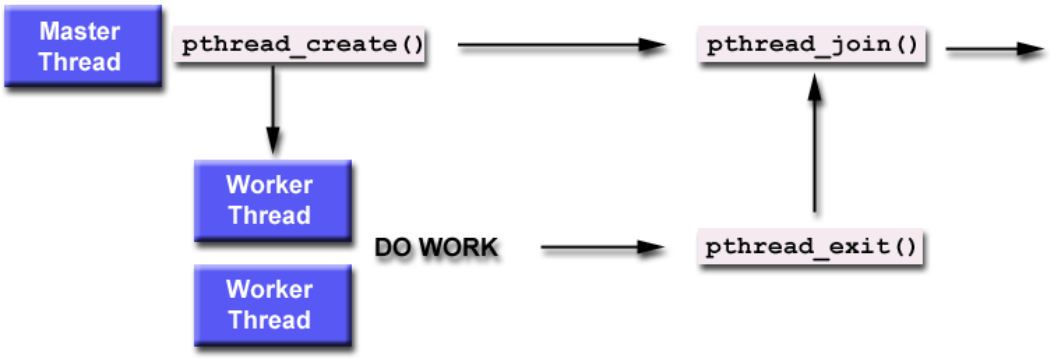
A thread is a separate flow of execution in a program. A single program can have multiple threads running within the same address space. Multithreaded programs share data in a common address space. Access to shared data is mediated through mutex locks.

Mutex locks have an owner and a queue of waiting threads. The initial state of a mutex lock is unlocked. Unlocked mutex locks have no owner. Once locked, a mutex can only be unlocked by its owner.

Multiple locks locked in the wrong order can cause deadlock. Lock all locks in a canonical locking order.

Thread Operations:

- Create -- pthread_create(thread, attr, start_routine, arg)
- Join -- A synchronization method. pthread_join() subroutine blocks the calling thread until the specified threadID thread terminates. It is a logical error to attempt multiple joins on the same threads



- Detach -- pthread_detach() routine is used to explicitly detach a thread even if it was created as joinable.
- Yield -- causes the calling thread to relinquish the CPU. The thread is placed at the end of the run queue for its static priority and another thread is scheduled to run
- Exit -- terminates the calling thread and makes the value value_ptr available to any successful join with the terminating thread. An implicit call to pthread_exit() is made when a thread other than the thread in which main() was first invoked returns from the start routine that was used to create it. The function's return value shall serve as the thread's exit status.

Creating a thread mallocs memory, passes it the the thread, and passes function. The thread executes function and frees memory.

****Never Kill a Thread****
Threads are more lightweight than processes and can be created with less system overhead.

Shell Scripting

Syntax:
$$ = PID
$# = Number of arguments
$? = Return value of last command called
$1 = return the first argument of script
$0 = name of script run
$@ = all arguments except 0
$* = all arguments
1>&2 = redirect stdout to stderror
&> = redirect both to a certain file

[-e FILE] True if FILE exists
[-d FILE] True if FILE exists and is directory
[-f FILE] True if FILE exists and is regular file
[-s FILE] True if FILE exists and size > 0
[-x FILE] True if FILE exists and is executable
[-n STRING] True if STRING length **n**onzero
[-z STRING] True if STRING length **z**ero

Conditionals -- if test_commands; then consequent_commands; fi

Example:

```
echo "checking string equality"

if [ -f ~/Code/tutorials/bashScripting/if.sh ]
   then
      echo "file exists"
fi
echo
echo "done"
```

Looping

Example

ls *.xml
output: file1.xml file2.xml file3.xml

ls *.xml > list

for i in `cat list` ; do cp “$i” “$i.bak” ; done

Shared Memory

Multiple processes share memory.
Adv: Speedy, no library or sys calls **Dis**: Still need a mutex/semaphore
Have a lifetime independent of any process. Larger than a process but smaller than a file.
Have Owners permissions.
No structure, just flat list with a key.
Different processes may attach to different addresses. Same program, same shared memory, different addresses.
int shmget(key_t, size_t, int flags)//returns shmID -- gets us new shared memory

int shmctl(int shmid, int cmd,struct shmid_ds *buf) /* performs the control operation specified by *cmd* on the shared memory segment whose identifier is given in *shmid*. */

int shmat(int shmid, const void * shmaddr, int shmflags) /* attaches the shared memory segment identified by *shmid* to the address space of the calling process */

int shmdt(void * shmaddr) /* detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process. The to-be-detached segment must be currently attached with *shmaddr* equal to the value returned by the attaching shmat() call */

Pointers to functions

typedef int (*CompareFuncT)(void *, void*);
CompareFuncT cf = compareInts; // same as &compareInts

Deadlock

4 Conditions Necessary for Deadlock // put this on cheat sheet!

- **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
- **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
- **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
- **Circular Wait** - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)]. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

Source: http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html

Semaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int counter);
```

```
// pshared == 0: the semaphore is local to the current process
// pshared != 0: the semaphore may be shared between processes
int sem_destroy(sem_t * sem);
sem_wait(sem_t *);      // lock
// counter == 0: suspend in semaphore queue
sem_trywait(sem_t *);   // trylock
// counter != 0: decrement counter by 1 and return immediately
// counter == 0: return -1 immediately – does not block
sem_post(sem_t *);     // unlock
```

Mutex vs Semaphore

- A mutex can only be unlocked by the thread that locked it
- Any thread can post to a semaphore
- A mutex has only two states: locked or unlocked
- A semaphore counter can take any nonnegative value
- A mutex can have at most one owner
- A semaphore can have multiple concurrent accessors

Good to know

Process vs Program vs Thread

```
execlp(const char *file, const char *arg0, ... /*, (char *)0 */);   // l is for list
execvp(const char *file, char *const argv[]);                       // v is for vector
// p in execlp and execvp means look in $PATH for *file
// *file argument contains '/' -> pathname (e.g. '/usr/bin/') , if no '/' -> uses $PATH
to find executable

execlp("sh", "sh", "-c", commandstring, NULL);    // equivalent to execvp() below
char *args[] = {"sh", "-c", commandstring, NULL}; // must terminate args with NULL ptr
execvp("sh", args);
```

The base of each is **exec** (execute), followed by one or more letters:

- **e** – An array of pointers to environment variables is explicitly passed to the new process image.
- **l** – Command-line arguments are passed individually to the function.
- **p** – Uses the PATH environment variable to find the file named in the *path* argument to be executed.
- **v** – Command-line arguments are passed to the function as an array of pointers.

User-level vs Kernel-level Threads

User-level threads run in user code. The mechanics of thread creation, destruction, scheduling (changing from thread to thread), etc run as (library) code in the user program without kernel support.

Pros: - Faster than kernel threads for CPU-intensive threads.
Cons: - kernel is unaware of multiple user threads. If one user thread gets blocked, all user thread stops.
- Can't schedule user-level threads on different processors.

Kernel-level threads run the mechanics in the kernel, not the user program. All thread operations, including creation, destruction, scheduling involve system calls to the kernel.

Pros: - One blocked kernel-level thread does not block other kernel-level threads.
- Faster for I/O-intensive threads.
- Can schedule kernel-level threads on different processors.
Cons: - Slower to create and manage than user-level threads (but not a lot slower).

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

// Function taking a function pointer as an argument

double compute_sum(double (*funcp)(double), double lo, double hi)
{
        double sum = 0.0;

        // Add values returned by the pointed-to function '*funcp'
        int i;
        for (i = 0; i < 1000; i++)
        {
                double x, y;

                // Use the function pointer 'funcp' to invoke the function
                x = i/1000.0 * (hi - lo) + lo;
                y = (*funcp)(x);
                sum += y;
        }
        return (sum/1000.0);
```

```
}

int main(void)
{
        double (*fp)(double);   // Function pointer
        double sum;

        // Use 'sin()' as the pointed-to function
        fp = sin;
        sum = compute_sum(fp, 0.0, 1.0);
        printf("sum(sin): %f\n", sum);

        // Use 'cos()' as the pointed-to function
        fp = cos;
        sum = compute_sum(fp, 0.0, 1.0);
        printf("sum(cos): %f\n", sum);
        return 0;
}
```