Chapter 11
Thread Control

Cheng-Hsin Hsu
National Tsing Hua University
Department of Computer Science

Parts of the course materials are courtesy of Prof. Chun-Ying Huang

Outline

- Introduction
- Thread limitations
- Thread attributes
- Synchronization attributes
- Thread-specific data
- Cancel options
- Threads and signals
- Threads and fork

Introduction

- We often use default settings for thread functions
- A NULL parameter is used for many thread function parameters
- For example:

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

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);

int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
```

Thread Limitations

Name of limit	Description	name argument (sysconf)
PTHREAD_DESTRUCTOR_ITERATIONS	max number of times an implementation will try to destroy the thread-specific data when a thread exits	_SC_THREAD_DESTRUCTOR_ITERATIONS
PTHREAD_KEYS_MAX	max number of keys that can be created by a process	_SC_THREAD_KEYS_MAX
PTHREAD_STACK_MIN	min number of bytes that can be used for a thread's stack	_SC_THREAD_STACK_MIN
PTHREAD_THREADS_MAX	max number of threads that can be created in a process	_SC_THREAD_THREADS_MAX

- Limitations can be obtained using sysconf function
  - `long sysconf(int name);`

Thread Limitations (Cont'd)

- Example of thread configuration limits

	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
PTHREAD_DESTRUCTOR_ITERATIONS	4	4	4	No limit
PTHREAD_KEYS_MAX	256	1024	512	No limit
PTHREAD_STACK_MIN	2048	16384	8192	8192
PTHREAD_THREADS_MAX	No limit	No limit	No limit	No limit

Thread Attributes

- The pthread_attr_t data type: Initialization and deinitialization

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- Common thread attributes

Name	Description	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
detachstate	detached thread attribute	•	•	•	•
guardsize	guard buffer size in bytes at end of thread stack	•	•	•	•
stackaddr	lowest address of thread stack	•	•	•	•
stacksize	lowest address of thread stack	•	•	•	•

detachstate

- We have introduced pthread_detach
- A thread can be in the state of detached or joinable
- We can set the thread detach state upon the creation of a thread
  - PTHREAD_CREATE_DETACHED
  - PTHREAD_CREATE_JOINABLE

```
int pthread_attr_getdetachstate(
            const pthread_attr_t *attr, int *detachstate);

int pthread_attr_setdetachstate(
            pthread_attr_t *attr, int detachstate);
```

- Returns zero on success, or non-zero error codes

Example: Create a Thread in Detached State

```
int
makethread(void *(*fn)(void *), void *arg) {
    int err;
    pthread_t tid;
    pthread_attr_t attr;
    err = pthread_attr_init(&attr);
    if (err != 0)
        return(err);
    err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (err == 0)
        err = pthread_create(&tid, &attr, fn, arg);
    pthread_attr_destroy(&attr);
    return(err);
}
```

Thread Stack Address and Size

- You may want to allocate memory for thread stack
  - The shared stack may be insufficient
  - Use memory spaces allocated by using malloc or mmap

```
int pthread_attr_getstack(const pthread_attr_t *attr,
            void **stackaddr, size_t *stacksize);
int pthread_attr_setstack(const pthread_attr_t *attr,
            void *stackaddr, size_t *stacksize);
```

  - Returns zero on success, or non-zero error codes
- The stackaddr parameter is the lowest addressable address in the range of memory – It is not necessarily the start of the stack
  - Stacks may grow from higher addresses to lower addresses, or
  - from lower addresses to higher addresses

Thread Stack Address and Size (Cont'd)

- We have pthread_attr_getstackaddr and pthread_attr_setstackaddr functions, but the use of these two functions are not recommended: In fact, the two functions are considered as deprecated
  - The stackaddr might be the beginning of the stack, or the lowest address of the stack ← may lead to unnecessary complications…
- We can also get or set the thread stack size

```
int pthread_attr_getstacksize(
            const pthread_attr_t *attr,
            size_t *stacksize);

int pthread_attr_setstacksize(
            pthread_attr_t *attr , size_t stacksize);
```

  - Return zero on success, or non-zero error codes

guardsize

- To protect stack overflow caused by a single thread
- There is a buffer at the end of a stack
- By default, the size is set to PAGESIZE bytes
- This feature can be disabled if the size is set to zero
- If a thread stack overflows, the process will receive an error, possibly with a unique signal – But actually you may simply get a SIGSEGV

```
int pthread_attr_getguardsize(const pthread_attr_t *attr,
                              size_t *guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr,
                              size_t guardsize);
```

  – Return zero on success, or non-zero error codes

Synchronization Attributes

- Synchronization attributes are used by mutexes, reader-writer locks, and condition variables
- All of them have similar initialization and destroy functions

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t
*attr);

int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

Mutex Attribute:Process-Shared

- By default, only threads in the same process can share the same mutex
- A mutex can be between processes
  - For example, we have shared memory mechanism
  - The process-shard attribute must be enabled

```
int pthread_mutexattr_getpshared(
                      const pthread_mutexattr_t *attr,
                      int *pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                 int pshared);
```

- The pshared value
  - PTHREAD_PROCESS_PRIVATE – More efficient implementation
  - PTHREAD_PROCESS_SHARED – More expensive implementation

Mutex Attribute: Type

- We have four exclusive types of mutex
  - PTHREAD_MUTEX_NORMAL
    - Standard mutex type that does not do any special error checking or deadlock detection
  - PTHREAD_MUTEX_ERRORCHECK
    - Provide error checking ← avoid: (i) double lock, (ii) double unlock, and (iii) unlock a mutex locked by another thread
  - PTHREAD_MUTEX_RECURSIVE
    - Allow the same thread to lock the mutex multiple times. The locker has to perform the same number of unlocks to release the mutex
  - PTHREAD_MUTEX_DEFAULT
    - The system dependent default choice of mutex type

Mutex Attribute: Type (Cont'd)

- Comparison of mutex type behavior

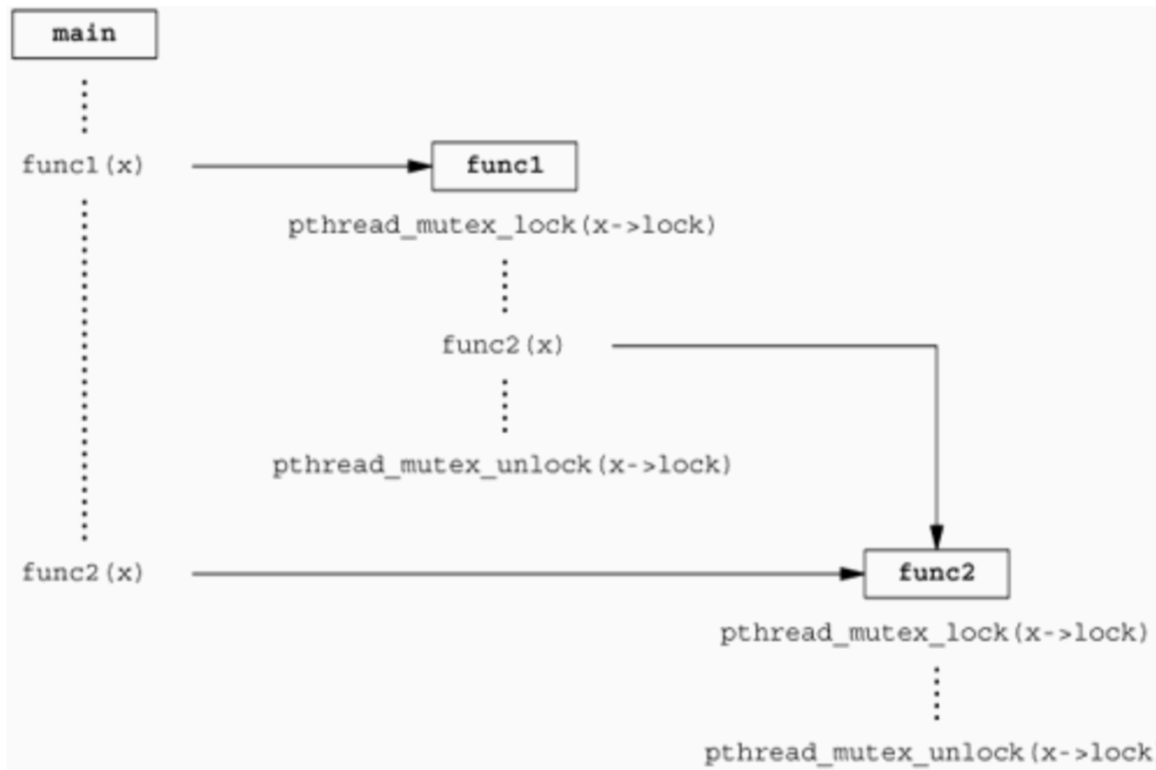
Mutex type	Relock without unlock?	Unlock when not owned?	Unlock when unlocked?
PTHREAD_MUTEX_NORMAL	deadlock	undefined	undefined
PTHREAD_MUTEX_ERRORCHECK	returns error	returns error	returns error
PTHREAD_MUTEX_RECURSIVE	allowed	returns error	returns error
PTHREAD_MUTEX_DEFAULT	system dependent	system dependent	system dependent

- Functions to get and get mutex type

```
int pthread_mutexattr_gettype(
                          const pthread_mutexattr_t * attr,
                          int *type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr,
                              int type);
```

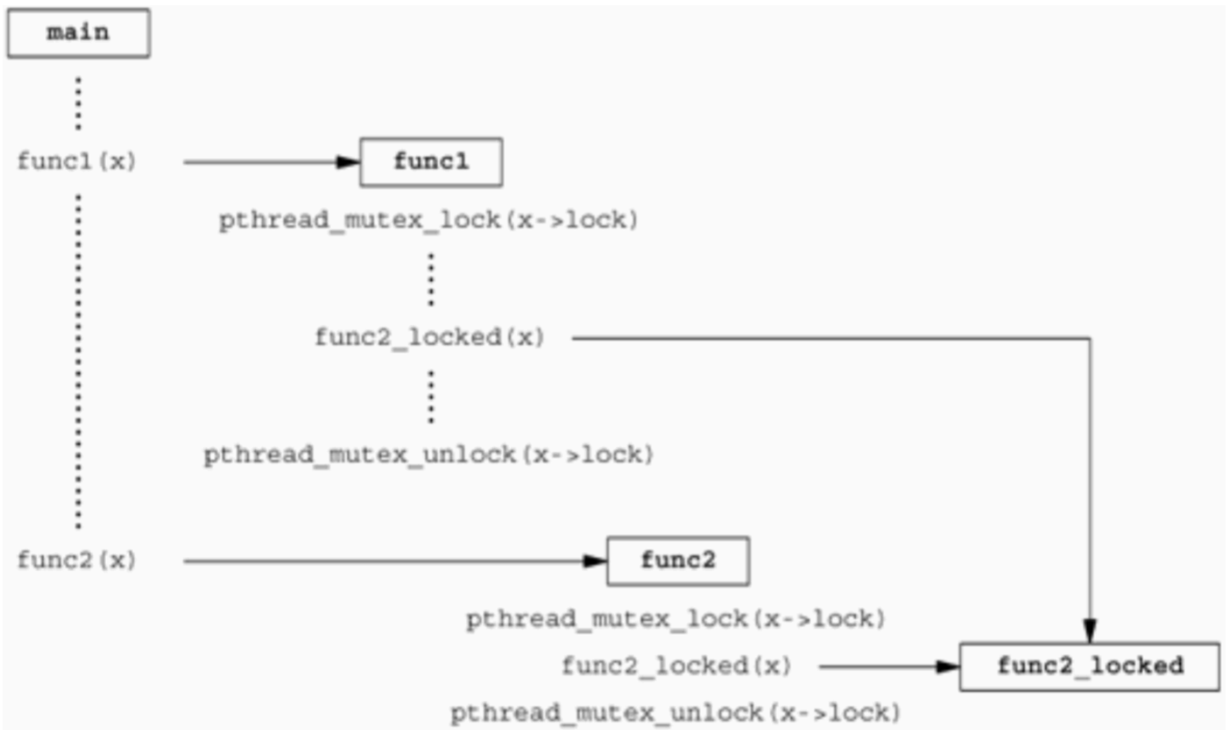
PTHREAD_MUTEX_RECURSIVE – A Common Scenario

- Assume we cannot modify func1 and func2
- Suppose func1 and func2 always try to lock an object
- If func1 calls func2 internally, there must be a deadlock
- A recursive mutex would prevent the deadlock in the scenario



PTHREAD_MUTEX_RECURSIVE – A Common Scenario (Cont'd)

- Another alternative to solve the same scenario
- Assume we are able to modify the codes
- We have two variants for func2:
  - A public version that locks the object
  - An internal version that does not lock the object
- func1 locks the object and calls the internal version



Other Common Attributes

- Reader-writer locks, condition variables, and barriers support process-shared attribute

```
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr,
                                  int *pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                  int pshared);

int pthread_condattr_getpshared(const pthread_condattr_t *attr,
                                int *pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr,
                                int pshared);

int pthread_barrierattr_getpshared(
                  const pthread_barrierattr_t *attr, int *pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
                                   int pshared);
```

Thread-Specific Data

- Thread-specific data, also called thread-private data
- We would like each thread to access its own separate copy of the data
- We do not have to worry about synchronizing access with other threads
- An straightforward solution
  – Use an array to store thread-specific data based on thread id
  – However, thread ids may be small (and incrementing) integers
  – Even if we have such an array, we still need extra protections to prevent a thread from accessing other threads' data
- Thread-specific data can be used to provide a mechanism for adapting process-based interfaces to a multithreaded environment
  – The errno example

Thread-Specific Data: Steps

- Create a pthread key – This should be done only ONCE for all threads in the same process
- Get the data associated with the key for the current thread
- If data is not available, allocate the data and associate the data with the key
- If data is no longer required, it can be released and de-associated

pthread Key

- Create and delete of thread key

```
int pthread_key_create(
                pthread_key_t *keyp, void (*destructor)(void *));

int pthread_key_delete(pthread_key_t *key);
```

- Before allocating thread-specific data, we need to create a key to associate with the data
- An optional destructor can be provided to release the data address when a thread exits
  - The non-NULL data address will be passed to the destructor
- A pthread key should be created only once
- A call to pthread_key_delete will NOT invoke the corresponding destructor

Example: Create a pthread Key

```
void destructor(void *);

pthread_key_t key;
int init_done = 0;

int threadfunc(void *arg) {
    if (!init_done) {
        init_done = 1;
        err = pthread_key_create(&key, destructor);
    }
    ...
}
```

- However, race conditions may happen for the blue lines
- We need a better solution

Example: Create a pthread Key (Revised)

- We can work with pthread_once function

```
pthread_once_t initflag = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *initflag, void (*initfn)(void));
```

```
void destructor(void *);

pthread_key_t key;
pthread_once_t init_done = PTHREAD_ONCE_INIT;

void thread_init(void) {
     err = pthread_key_create(&key, destructor);
 }

int threadfunc(void *arg) {
     pthread_once(&init_done, thread_init);
    ...
}
```

Get, Associate, and De-associate Data

- Get

```
void *pthread_getspecific(pthread_key_t key);
```

  - Return non-NULL for the associated value, or NULL if no value has been associated with the key
- Associate and de-associate

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

  - Use a non-NULL value to associate the data
  - Use a NULL data to de-associate the data, previously associated data should be retrieved and released first
  - Return zero on success, or non-error error codes

Example: A Thread-Safe Implementation of getenv

```
static pthread_key_t key;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;
pthread_mutex_t env_mutex = PTHREAD_MUTEX_INITIALIZER;

extern char **environ;

static void thread_init(void) {
    pthread_key_create(&key, free);
}

char * getenv(const char *name) {
    int i, len;
    char *envbuf;
    pthread_once(&init_done, thread_init);
```

Example: A Thread-Safe Implementation of getenv (Cont'd)

```
    pthread_mutex_lock(&env_mutex);
    envbuf = (char *) pthread_getspecific(key);
    if (envbuf == NULL) {
        if((envbuf = malloc(ARG_MAX)) == NULL) {
    pthread_mutex_unlock(&env_mutex);
            return(NULL);
        }
        pthread_setspecific(key, envbuf);
     }
     len = strlen(name);
     for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0)
        &&  (environ[i][len] == '=')) {
            strcpy(envbuf, &environ[i][len+1]);
            pthread_mutex_unlock(&env_mutex);
            return(envbuf);
        }
    }
    pthread_mutex_unlock(&env_mutex);
    return(NULL);
}
```

Cancel Options: Cancel State

- Recall that the pthrad_cancel function simply send a "cancellation request" to the target thread
- The caller of pthread_cancel does not wait for thread termination
- The target thread may be not terminate immediately
- The target thread is terminated at a "cancellation point"
- We can temporarily disable "cancellation points"
  - If we have some critical codes that must not be interrupted by cancellation requests
- We can setup the "cancel state"

Cancel Options: Cancel State (Cont'd)

- The cancel option is not included in the pthread attribute

```
int pthread_setcancelstate(int state, int *oldstate);
```

  - Return: zero on success, or non-zero error codes
- The cancelability can be:
  - PTHREAD_CANCEL_ENABLE (the default)
  - PTHREAD_CANCEL_DISABLE
- List of cancellation points are shown in the next slide
- If a thread does not call any of the cancellation point functions, by default it will not be terminated
- You can manually embed cancellation point in your program

```
void pthread_testcancel(void);
```

  - pthread_testcancel also not works when the cancel option is set to DISABLED

List of Cancellation Points

- Defined by POSIX.1
- There are also cancellation points optionally defined by POSIX.1 (omitted, please refer to the text book)

accept	mq_timedsend	putpmsg	sigsuspend
aio_suspend	msgrcv	pwrite	
sigtimedwait			
clock_nanosleep	msgsnd	read	sigwait
close	msync	readv	sigwaitinfo
connect	nanosleep	recv	sleep
creat	open	recvfrom	system
fcntl2	pause	recvmsg	tcdrain
fsync	poll	select	usleep
getmsg	pread	sem_timedwait	wait
getpmsg	pthread_cond_timedwait	sem_wait	waitid
lockf	pthread_cond_wait	send	waitpid
mq_receive	pthread_join	sendmsg	write
mq_send	pthread_testcancel	sendto	writev
mq_timedreceive	putmsg	sigpause	

Cancel Options: Cancel Type

- We have mentioned that a thread is cancelled at cancel points
- So the cancellation of a thread is deferred to a cancel point
- If we would like a thread to be cancelled immediately, we can change the cancel type

```
int pthread_setcanceltype(int type, int *oldtype);
```

- The type can be
  - PTHREAD_CANCEL_DEFERRED (the default)
  - PTHREAD_CANCEL_ASYNCHRONOUS
- If the cancel state is set to DISABLED, a thread will be not cancelled

Threads and Signals

- The signal disposition is shared by all threads
- But each thread has their own signal mask
- Signals are delivered to only one thread in the process
  - If the signal is related to a hardware fault or expiring timer, the signal is sent to the thread whose action caused the event
  - Other signals are delivered to an arbitrary thread
  - So usually we block unused signals in threads, and prevent signals from being sent to an incorrect thread
- Setting up per-thread signal mask
  - The parameters are equivalent to sigprocmask function
  - You have to use pthread_sigmask instead of sigprocmask

```
int pthread_sigmask(int how, const sigset_t *set,
                                   sigset_t *oset);
```

Thread: Wait for a Signal

- A thread is able to wait for a signal using sigwait function

```
int sigwait(const sigset_t *set, int *signop);
```

  - The set argument specifies the signals to wait
  - The signop stores the number of signal that was delivered
- Usually we have to block signals that will be waited by sigwait
- sigwait atomatically unblocks signals and wait until one is delivered
- Multiple signal receivers
  - If a thread has registered a signal handler as well as made function call to sigwait, only one (the handler or sigwait) will receive the signal – that is implementation dependent
  - If two threads calls sigwait to wait for the same signal, only one will receive the signal

Send a Signal to a Thread

- Similar to kill, we can send a signal to a thread

```
int pthread_kill(pthread_t thread, int signo);
```

  – Return zero on success, or non-zero error codes
- We may pass a value zero to signo to check the existence of a thread
- If a default signal action for a signal is to terminate the process, the entire process will be killed

Threads and fork()

- A child process inherits a lot from its parent
- Include mutex, reader-writer lock, and condition variables
- In a multi-threaded program, only ONE thread is in the child process
  - That's the thread calls fork
- Locks held by other threads will be NOT released, and there is no way for the child thread to release the locks
- The lock problem will not happen if a child process calls exec
  - All the old address space is discarded, so the lock state doesn't matter
- How to avoid such a problem if a child process does not call exec?

The pthread_atfork Function

- Prototype

```
int pthread_atfork(void (*prepare)(void),
                   void (*parent)(void), void (*child)(void));
```

  – Return zero on success, or non-zero error codes
- The prepare function is called before fork() function is executed
- The parent function is called after fork() @ the parent process
- The child function is called after fork() @ the child process

Solution to the Lock Problem

- Acquire all the locks in the parent function
- This is to guarantee that all the locks have been unlocked and then acquired by the parent function before fork is performed
- Unlock the locks in both the parent function and the child function, so the lock states at the parent and the child are synchronized (all are unlocked)

Example of the Lock Problem: The Worker

- Lock, sleep for 3 seconds, and then unlock

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *worker(void *arg) {
        pthread_mutex_lock(&lock);
        puts("worker: locked.");
        sleep(3);
        puts("worker: unlocked.");
        pthread_mutex_unlock(&lock);
        return(0);
}
```

Example of the Lock Problem: The Parent and the Child

```
int main(void) {
        pid_t pid;
        pthread_t tid;

    pthread_create(&tid, NULL, worker, 0);
        sleep(1);
        puts("parent: The lock is held by the worker thread.");
    if ((pid = fork()) == 0) {
                puts("child: start.");  // the only thread @
child
                pthread_mutex_lock(&lock);
                puts("child: locked."); // never reach here
                pthread_mutex_unlock(&lock);
                puts("child: terminated.");
                return 0;
        }
    pthread_join(tid, NULL);
    return 0;
}
```

Example of the Lock Problem: The Callback Functions

```
void prepare(void) {
        pthread_mutex_lock(&lock);
}

void parent(void) {
        pthread_mutex_unlock(&lock);
}

void child(void) {
        pthread_mutex_unlock(&lock);
}
```

Example of the Lock Problem: Revised Codes

```
int main(void) {
        pid_t pid;
        pthread_t tid;
    pthread_atfork(prepare, parent, child);
    pthread_create(&tid, NULL, worker, 0);
        sleep(1);
        puts("parent: The lock is held by the worker thread.");
    if ((pid = fork()) == 0) {
                puts("child: start.");  // the only thread @
child
                pthread_mutex_lock(&lock);
                puts("child: locked."); // lock ok
                pthread_mutex_unlock(&lock);
                puts("child: terminated.");
                return 0;
        }
    pthread_join(tid, NULL);
    return 0;
}
```

Assignment #9 (5%)

1. (2%) Given that you can create multiple threads to perform different tasks within a program, explain why you might still need to use fork.
2. (3%) After calling fork, could we safely reinitialize a condition variable in the child process by first destroying the condition variable with pthread_cond_destroy and then initializing it with pthread_cond_init?