CSC 374/407: Computer Systems II

Lecture 5

Joseph Phillips
De Paul University

2014 August 5

Reading

- Bryant & O'Hallaron "*Computer Systems, 2nd Ed.*"
  - Chapter 13.1-13.5: Concurrent Programming
- Hoover "*System Programming*"
  - (None)

Topics

- What threads are and why use them
- How to create threads
- Critical sections and unsafe thread programming
- Synchronization
- Application: Producer/Consumer

What threads are and why use them

Purpose: to have multiple “little processes” parallelize action of one program

Threads are like processes, but they share:

- Code segment
- Heap segment
- Data/BSS segments
- Stack segment

They differ in that:

- Have different registers (including condition codes)
- Start in different regions of stack
- Have unique thread id

How to compile for threads

At the beginning of your program:

```
. . .
#include <thisHeader.h>
#include <pthread.h>
#include <thatHeader.h>
. . .
```

When you link:

```
unix> gcc . . . -lpthread . . .
```

Why are they called “p-threads?”

- ***P*** = ***POSIX*** = ***P***ortable ***O***perating ***S***ys ***I***nterface for Uni***X***
- Standardizes “Unix” across Linux, BSD, Solaris, *etc*.

How to create threads

```
int pthread_create
    (pthread_t* restrict           thread,
     /* Pointer to a pthread_t object to identify
        the child thread */
     const pthread_attr_t* restrict  attr,
     /* Pointer to optional object for properties
        of child.  You can just say NULL. */
     void *(*fncName)(void*),
     /* Name of function to run:
        void* fncName(void* ptr) */
     void *restrict                 arg
     /* Ptr to object that is arg to fncName() */
    )
```

Return value is:

- 0: success
- anything else: ERROR!

How to wait for threads

```
int pthread_join
        (pthread_t              thread,
            /* Thread for which to wait */
         void**               valuePtr
            /* Value returned by pthread_exit()
            pointed to by valuePtr (may be NULL)
            */
        )
```

Compare threads with fork:

```
/* I use fork() to create a process */

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
 int     childsStatus;
 pid_t childId      = fork();
 char* argsText  = /* whatever */

 if  (childId == 0)
    execlp(“doThisFile”,..,argText,.. );

 waitpid(childId,&childStatus,0);

 return(0);
}
```

```
/* I use pthread_create() */

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

int main()
{
 pthread_t            childId;
 SendObject          arg = /*whatever*/
 ReceiveObject*   childsStatusPtr;

 pthread_create(&childId,NULL,
                         doThisFnc,&arg
                        );

 pthread_join(childId,&childStatusPtr);
 return(0);
}
```

Example program:

```
/* Compile with:
 * gcc -lpthread thread_ex1.c -o thread_ex1
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
const int N = 2;

/* thread routine */
void *thread_routine(void *vargp)
{
    int id = *(int*)vargp;
    char* ptr;
    printf("Hello from child thread %d\n", id);
    switch (id)
        {
      case 0:     ptr = strdup("Hello "); break;
      case 1:     ptr = strdup("there!"); break;
        }
    return(ptr);
}
```

```
int main()
{
    int          i;
    char*        msgPtr;
    pthread_t    tid[N];

    for (i = 0; i < N; i++)
        pthread_create
            (&tid[i], NULL,
             thread_routine, (void *)&i);

    for (i = 0; i < N; i++)
        {
        pthread_join(tid[i], (void**)&msgPtr);
        puts(msgPtr);
        free(msgPtr);
        }

    return(0);
}
```

Time for you!

Can you write a threaded program that can:

1. **Prove** that all threads use the same **stack** (did we just do that?)
2. **Prove** that all threads use the same **global var data space** (for global vars and static vars inside functions)
3. **Prove** that all threads use the same **heap**.

What's wrong with this?

```
/* Compile with:
 * gcc -lpthread badcnt.c -o badcnt
 */
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

unsigned int cnt = 0; /* shared */
unsigned int NUM_ITERS;

/* thread routine */
void *count(void *arg)
{
    int i;
    for (i=0; i<NUM_ITERS; i++)
        cnt++;
    return NULL;
}
```

```
int main(int argc, char* argv[])
{
    pthread_t tid1, tid2;
    if  ( (argc >= 2)  &&  isdigit(*argv[1]) )
        NUM_ITERS = atoi(argv[1]);
      else
        {
        const int LINE_SIZE = 16;
        char line[LINE_SIZE];
        do  {
            printf("How many iterations? ");
            fgets(line,LINE_SIZE,stdin);
            }
        while  ( !isdigit(line[0]) );
        NUM_ITERS = atoi(line);
        }
    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf(“Should be %d is %d\n”,NUM_ITERS*2,cnt);
    return(0);
}
```

Output:

```
[jphillips@localhost]$  badcnt 10
Should be 20 is 20
[jphillips@localhost]$  badcnt 100
Should be 200 is 200
[jphillips@localhost]$  badcnt 1000
Should be 2000 is 2000
[jphillips@localhost]$  badcnt 10000
Should be 20000 is 20000
[jphillips@localhost]$  badcnt 100000
Should be 200000 is 200000
[jphillips@localhost]$  badcnt 1000000
Should be 2000000 is 1830352
[jphillips@localhost]$  badcnt 10000000
Should be 20000000 is 15214384
```

Last two values are WRONG!

Critical sections

```
(gdb) disass count
0x400688:   push   %rbp
0x400689:   mov    %rsp,%rbp
0x40068c:   mov    %rdi,0xffffffffffffffe8(%rbp)
0x400690:   movl   $0x0,0xfffffffffffffffc(%rbp)
0x400697:  jmp    0x4006ac <count+36>
0x400699:  mov    2098629(%rip),%eax   # 0x600c64 <cnt>
0x40069f:  add    $0x1,%eax
0x4006a2:  mov    %eax,2098620(%rip)   # 0x600c64 <cnt>
0x4006a8:  addl   $0x1,0xfffffffffffffffc(%rbp)
0x4006ac:  mov    0xfffffffffffffffc(%rbp),%edx
0x4006af:  mov    2098611(%rip),%eax  #0x600c68 <NUM_ITERS>
0x4006b5:  cmp    %eax,%edx
0x4006b7:  jb     0x400699 <count+17>
0x4006b9:  mov    $0x0,%eax
0x4006be:  leaveq
0x4006bf:  retq
```

These three must be done by one thread at a time!

Critical sections (2)

Imagine some scenarios where they are **not** done atomically

Other ways threads can step on each other's toes

1. Working with the same global variable
   - (Just covered)
2. Working with the same static var within a fnc.
   - Can *you* think of an example?
3. A function returning a pointer to a static var.
   - Can *you* think of an example?
4. Calling a thread-unsafe function.
   - Can *you* think of an example?

Functions: safe and unsafe

Reentrant functions call no shared variables

- Always thread safe

Most standard C Library functions are thread safe

- *e.g*. printf(), malloc()

Unsafe to safe

– asctime() (***unsafe!***)	asctime_r() (***safe!***)
– ctime() (***unsafe!***)	ctime_r() (***safe!***)
– gethostbyaddr() (***unsafe!***)	gethostbyaddr_r()(***safe!***)
– inet_ntoa(***unsafe!***)	***NO SAFE VERSION!***
– localtime() (***unsafe!***)	localtime_r() (***safe!***)
– rand() (***unsafe!***)	rand_r() (***safe!***)

The semaphore solution

Classic solution: Dijkstra's P and V operations on *semaphores.*

- *semaphore:* non-negative integer synchronization variable.
  - sem_wait(s): [`while(s==0) wait(); s--;`]
    - Originally named P(), Dutch for "Proberen" (test)
  - sem_post(s): [`s++;`]
    - Originally named V(), Dutch for "Verhogen" (increment)
- OS guarantees that operations between brackets [ ] are executed indivisibly.
  - Only one P or V operation at a time can modify s.
  - When `while` loop in P terminates, only that P can decrement s.

Semaphore invariant: *(s >= 0)*

(1) POSIX Semaphores

What to include:

- <semaphore.h> preceded by #include

Types and functions:

- sem_t semaphore;
- sem_init(sem_t* semPtr, int flag, int value)
  - Initialize pointed-to semaphore, with value, if flag == 1 then semaphore can be forked
- sem_destroy(sem_t* semPtr)
  - Destroy pointed-to semaphore. If it's negative then block.

POSIX Semaphores, cont'd

- sem_wait(sem_t* semPtr)
  - Decrement pointed-to semaphore. If it's negative then block.
- sem_post(sem_t* semPtr)
  - Increment pointed-to semaphore.   Wake one blocked process if any.
- sem_getvalue(sem_t* semPtr, int* valuePtr)
  - Get value of pointed to semaphore.

POSIX semaphore solution

```
sem_t      sem;

/* P operation on semaphore sem */
void P(sem_t *semPtr)  {
  if (sem_wait(semPtr))  {
    fputs("sem_wait error\n",stderr);
    exit(-1);
    }
}

/* V operation on semaphore sem */
void V(sem_t *semPtr)  {
  if (sem_post(semPtr))  {
    fputs("sem_post error\n",stderr);
    exit(-1);
    }
}
```

```
/* thread routine */
void *count(void *arg)  {
    int i;
    for (i=0; i<NUM_ITERS; i++)   {
        P(&sem);
        cnt++;
        V(&sem);
        }
    return NULL;
}


/* in main */
. . .
/* initialize sem to 1 */
    if (sem_init(&sem, 0, 1) < 0)   {
        fputs("sem_init error\n",stderr);
        exit(1);
        }
```

(2) *pthread_mutex* solution

- pthread_mutex_t lock
- pthread_mutex_init(addressOfLock,NULL)
  - Makes a lock and initializes it to default values.
- pthread_mutex_destroy(addressOfLock)
  - Makes a lock and initializes it to default values.
- pthread_mutex_lock(address of lock)
  - Blocks thread until lock obtained, then obtains lock and blocks other threads until lock released.
- pthread_mutex_unlock(address of lock)
  - Releases lock allowing other threads to obtain it.

pthread_mutex solution

```
pthread_mutex_t cnt_lock;

/* thread routine */
void *count(void *arg)  {
    pthread_mutex_t* mutexLockPtr;
    mutexLockPtr = (pthread_mutex_t*)arg;

    for (int i=0; i<NUM_ITERS; i++) {
        pthread_mutex_lock(mutexLockPtr);
        cnt++;
        pthread_mutex_unlock(mutexLockPtr);
        }
    return NULL;
}

/* in main() */
. . .
pthread_mutex_init(&cnt_lock,NULL);
```

```
[jphillips@localhost]$ bettercnt 10
Should be 20 is 20
[jphillips@localhost]$ bettercnt 100
Should be 200 is 200
[jphillips@localhost]$ bettercnt 1000
Should be 2000 is 2000
[jphillips@localhost]$ bettercnt 10000
Should be 20000 is 20000
[jphillips@localhost]$ bettercnt 100000
Should be 200000 is 200000
[jphillips@localhost]$ bettercnt 1000000
Should be 2000000 is 2000000
[jphillips@localhost]$ bettercnt 10000000
Should be 20000000 is 20000000
[jphillips@localhost]$ bettercnt 100000000
Should be 200000000 is 200000000
```

Producer-Consumer

- One or more threads produce something
  - Place in buffer
- One or more threads consume something
  - Retrieve from buffer
- Critical section
  - Access of pointers/indices for buffer
- Problem!
  - A producer may gain access to a full buffer
  - A consumer may gain access to an empty buffer
- Oh no! An example in ***C++***!
  - A buffer is an ***object***.
  - Make the ***object*** thread-safe.
  - Lessons ***carry over to*** Java, C#, *etc*.

Unsafe_Buffer.h

```
class   Buffer
{
  enum { SIZE   = 16 };

  int   array_[SIZE];
  int   inIndex_;
  int   outIndex_;
  int   numItems_;

public :

  Buffer         ()
  {
    inIndex_ = outIndex_
      = numItems_ = 0;
  }

  ~Buffer        ()
  {
  }
```

```
int   getNumItems   () const
{ return(numItems_); }

void  putIn (int    i)
{
  while  (getNumItems() >= SIZE)
  {
    printf("Full!  Waiting!\n");
    usleep(10);
  }

  array_[inIndex_] = i;
  countArray[array_[inIndex_]]++;
  usleep(10 + rand() % 10);
  inIndex_++;
  numItems_++;
  if  (inIndex_ >= SIZE)
    inIndex_ = 0;
}
```

Unsafe_Buffer.h

```
  int   pullOut ()
  {
    while  (getNumItems() <= 0)
    {
      printf("Empty!  Waiting!\n");
      usleep(10);
    }

    countArray[array_[outIndex_]]--;
    int toReturn        = array_[outIndex_];
    usleep(10 + rand() % 10);

    outIndex_++;
    numItems_--;
    if  (outIndex_ >= SIZE)
      outIndex_ = 0;

    return(toReturn);
  }
};
```

producerConsumer.cpp

```
#include        <stdlib.h>
#include        <stdio.h>
#include        <unistd.h>
#include        <pthread.h>
int*            countArray;
#include        "Unsafe_Buffer.h"

const int       NUM_INTEGERS_TO_BUFFER  = 0x1000;

void*   stuffIntegersIn (void*  vPtr)
{
  for  (int i = 0;  i < NUM_INTEGERS_TO_BUFFER;  i++)
    ((Buffer*)vPtr)->putIn(i);

  return(NULL);
}
```

producerConsumer.cpp

```
void*   pullIntegersOut (void*  vPtr)
{
  for  (int i = 0;  i < NUM_INTEGERS_TO_BUFFER;  i++)
  {
    int j = ((Buffer*)vPtr)->pullOut();

    printf("Trial %d got %d.\n",i,j);
    fflush(stdout);
  }

  return(NULL);
}
```

producerConsumer.cpp

```
int     main    ()
{
  pthread_t     producer0;
  pthread_t     producer1;
  pthread_t     consumer0;
  pthread_t     consumer1;
  Buffer        buffer;
  countArray    =
  (int*)calloc(NUM_INTEGERS_TO_BUFFER,sizeof(int));

  pthread_create(&producer0,NULL,stuffIntegersIn,&buffer);
  pthread_create(&producer1,NULL,stuffIntegersIn,&buffer);
  pthread_create(&consumer0,NULL,pullIntegersOut,&buffer);
  pthread_create(&consumer1,NULL,pullIntegersOut,&buffer);
```

producerConsumer.cpp

```
  pthread_join(producer1,NULL);
  pthread_join(producer0,NULL);
  pthread_join(consumer1,NULL);
  pthread_join(consumer0,NULL);

  for  (int i = 0;  i < NUM_INTEGERS_TO_BUFFER;  i++)
    if  (countArray[i] < 0)
      printf("%d was gotten too many times!\n",i);
    else
    if  (countArray[i] > 0)
      printf("%d was put too many times!\n",i);

  return(EXIT_SUCCESS);
}
```

Your turn!

(1) Run it as-is. ***Any problems?***

Your turn!

(2) Make it thread-safe.

```
void myMethod ()
{
  pthread_mutex_lock(&lock);
  doCriticalSection();
  pthread_mutex_unlock(&lock);
}
```

NOTE:

1) Each object gets its ***own*** lock.
2) All threads accessing the same object use ***same*** lock

Any problems still?

Solution: *`pthread_cond`*!

- `pthread_cond_t     cond`
- `pthread_cond_init(addressOfCondition, NULL)`
  - Makes a condition and initializes it to default values.
- `pthread_cond_destroy(addressOfCondition)`
  - Destroys condition.
- `pthread_cond_wait(addrOfCondition, addrOfLock)`
  - Blocks thread until lock released and condition signaled. Then obtains lock again.
- `pthread_cond_signal(addressOfCondition)`
  - Signal *one waiting thread* that condition is met.
- `pthread_cond_broadcast(addressOfCondition)`
  - Signal *all waiting threads* that condition is met.

The ***proper*** solution

```
void myMethod ()
{
  pthread_mutex_lock(&lock);
  while ( !this->isReady() )
    pthread_cond_wait(&wCond,&lock);
  doCriticalSection();
  pthread_cond_signal(&sCond);
  pthread_mutex_unlock(&lock);
}
```

NOTE:

(1) Method **`isReady()`** determines if this object is ready for thread to do **`myMethod()`**.

(2) Conditions **`wCond`** and **`sCond`** may or may not be same.

We have mutual exclusion, are guaranteed to be safe?

Sorry but No!

- Race conditions
  - When result is dependent on order of processes or threads
- Deadlock
  - Thread/process A has lock 1 and is waiting for lock 2
  - Thread/process B has lock 2 and is waiting for lock 1

Next time: *Memory!*