Threads

COMP 321

Dave Johnson

1

Review: Processes and Threads

Classically, a process has a single thread of execution

- One point of execution progress, one set of register values
- Example:

```
main( … )
{
        …
        return 0
}
```

But a process may be "multithreaded"

- Multiple "***threads***" sharing the same address space
- All running concurrently, all "at once," cooperating
- Threads are also called ***lightweight processes***

2

Why Multiple Threads Sharing an Address Space?

Easy cooperation between these threads since they share all data, such as

- ***A windowing GUI system***
  - All threads share the same data structures of what's on the screen
  - One thread tracking the mouse on the screen
  - One thread for each open window
- ***Microsoft Word***
  - On thread managing the user's keyboard
  - One thread doing line breaks, one doing paragraph breaks, one page breaks
  - One thread doing spell checking, one for grammar checking, etc.

Creating a New Thread

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

Creates a new thread that begins by calling the function start_routine(arg)

- Puts the thread id of the new thread into memory at address "thread"
- The "attr" (attributes) is usually NULL, giving default thread attributes
- ***So to create a new thread that begins by calling my_func(my_arg), use***

```
pthread_t  tid;
int error;                  // the errno value is returned rather than put in errno
error = pthread_create(&tid,  NULL,  my_func,  my_arg);
```

A Simple pthread_create() Example

```
struct my_args  {  int a, b, c; }  data;

pthread_t  tid;
int error;

data.a = 3;
data.b = 2;
data.c = 1;

error = pthread_create(&tid,  NULL,
      my_thread,  &data);

. . .
```

```
void *
my_thread(void *arg)
{
      struct my_args *data = arg;

      printf("%d %d %d\n",
          data->a, data->b, data->c);

      . . .
}
```


fork() vs. pthread_create()

fork()	pthread_create()
creates a new process	creates a new thread in the ***same*** process
creates a new address space	new thread runs in the ***same*** address space
new process begins execution at the return from fork(), the same place as the parent	new thread begins execution by calling specified procedure; the parent returns normally from pthread_create()
no arguments are passed to the new process, since it isn't a procedure call	A single "generic" pointer argument is passed to the new thread

Each Thread Has Its Own Stack

Each thread executes concurrently and independently

- Each thread does its own procedure calls and returns
- Each thread has its own local variables
- Each thread has its own register values, including its own stack pointer

pthread_create() allocates the memory for the new thread's stack

- The size of the stack for a new thread
  - Normally, just use the default size (on CLEAR, 2 MB)
  - Can use pthread_attr_setstacksize() to set the stack size in the pthread_attr_t attributes that you will pass to pthread_create()
- The memory for the stack is freed automatically when the thread terminates

Termination of a Thread

```
[[noreturn]] void pthread_exit(void *retval);
```

A thread terminates when any of the following happens

- When the thread calls pthread_exit() – retval may be NULL
- Or when the thread returns from its "start routine" – this is the same as the thread calling pthread_exit() with that same pointer
- Or when it is canceled by some thread calling pthread_cancel() with its tid
- Or when ***any*** of the threads in that process call exit(3) or _exit() or the main thread returns from main() or the process otherwise terminates
  - ***Beware: this terminates all threads in that process***

Waiting (or Not) for a Thread to Terminate

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

Waits for the specified thread to terminate (must wait for a specific tid)

- If that thread has already terminated, pthread_join() returns immediately
- retval gives address of where to put a copy of retval pointer from the thread
  - If retval here is NULL, the return value from the thread is thrown away

```
int  pthread_detach(pthread_t  thread);
```

Changes the thread to "detached" so that it cleans up for itself on termination

- No one needs to (or can) call pthread_join() on it, no need to wait on it

Thread Synchronization

Remember our X = X + 1 and X = X + 2 example (and worse)

- The threads in a process all share the same memory (process's address space)
- Need a way to synchronize threads and allow them to cooperate safely

Pthreads provides two interrelated mechanisms

- ***mutex***
  - Allows threads to ensure that no more than one thread at a time is executing inside some critical section of the program's code
- ***condition variable*** (with an associated mutex)
  - Allows threads to do conditional behavior, waiting for some condition, coordinated and still protected by the mutual exclusion of associated mutex

Mutexes

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

- Must initialize it before use (can instead set to PTHREAD_MUTEX_INITIALIZER)
  - Use NULL for attr for default attributes
- If you dynamically allocate memory for mutex, must destroy it before free

```
int  pthread_mutex_lock(pthread_mutex_t  *mutex);
int  pthread_mutex_unlock(pthread_mutex_t  *mutex);
```

- Between lock and unlock, any other thread attempting lock will be blocked

A Simple Mutex Example

```
pthread_mutex_t  mutex;
pthread_mutex_init(&mutex, NULL);

int X = 0;
```

Thread 1

```
pthread_mutex_lock(&mutex);
X = X + 1;
pthread_mutex_unlock(&mutex);
```

Thread 2

```
pthread_mutex_lock(&mutex);
X = X + 2;
pthread_mutex_unlock(&mutex);
```

Using the mutex guarantees X = 3 when this is all done

Condition Variables

A condition variable has no actual value!

- It's a way to wait for or signal the occurrence of some generalized condition
- Internally, it maintains a list/set/collection of threads waiting on it
- But that is not visible and cannot be directly manipulated

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

- Must initialize it before use (can instead set to PTHREAD_COND_INITIALIZER)
  - Use NULL for attr for default attributes
- If you dynamically allocate memory for cond var, must destroy it before free

Condition Variables

Can only be used while you have the associated mutex locked

```
int  pthread_cond_wait(pthread_cond_t *restrict  cond,
         pthread_mutex_t *restrict  mutex);
```

- ***Atomically*** releases the associated mutex ***and*** adds this thread to those collection of those threads waiting on this condition
  - The actual condition (its meaning) depends on the code in how it is used

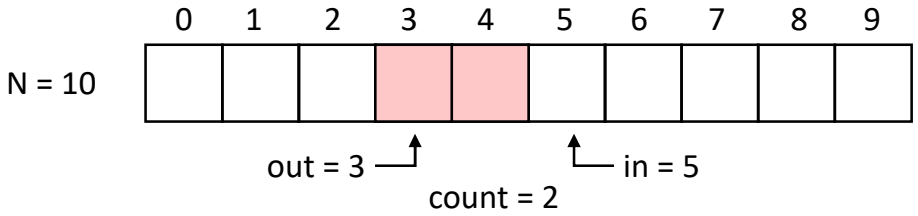
```
int  pthread_cond_signal(pthread_cond_t  *cond);
```

- Unblocks one (really, at least one) thread waiting on this condition variable
  - Which automatically ***relocks the mutex*** (when it can) before proceeding
- If no threads are waiting on this condition variable, the signal has ***no effect***

Example: The Bounded-Buffer Problem

Problem definition

- A ***buffer*** that can hold N "items"
- ***One producer*** thread that repeatedly produces an item and adds it to buffer
- ***One consumer*** thread that repeatedly removes an item from the buffer and consumes it



- (Consider alternate versions with ***more than*** one producer and one consumer)

15

A Solution to the Bounded-Buffer Problem

```
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t  full = PTHREAD_COND_INITIALIZER;
pthread_cond_t  empty = PTHREAD_COND_INITIALIZER;

struct item  buffer[N];
int  in = 0, out = 0, count = 0;

void add_item(struct item *data) {
        . . .
}

void remove_item(struct item *data) {
        . . .
}
```


16

A Solution to the Bounded-Buffer Problem

```
int main() {
    pthread_t  prod, cons;

    pthread_create(&prod,
        NULL, producer, NULL);
    pthread_create(&cons,
        NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    exit(0);
}
```

```
void *producer(void *arg) {
    struct item new;
    while (1) {
        . . .
        add_item(&new);
    }
}

void *consumer(void *arg) {
    struct item new;
    while (1) {
        remove_item(&new);
        . . .
    }
}
```


The add_item and remove_item Procedures

```
void add_item(struct item *data)
{
   pthread_mutex_lock(&mutex);

   while (count == N)
      pthread_cond_wait(&full, &mutex);

   buffer[in] = *data;
   count++;
   in = (in + 1) % N;

   pthread_cond_signal(&empty);

   pthread_mutex_unlock(&mutex);
}
```

```
void remove_item(struct item *data)
{
   pthread_mutex_lock(&mutex);

   while (count == 0)
      pthread_cond_wait(&empty, &mutex);

   *data = buffer[out];
   count--;
   out = (out + 1) % N;

   pthread_cond_signal(&full);

   pthread_mutex_unlock(&mutex);
}
```

Example: The Dining Philosophers Problem

A round table with 5 philosophers

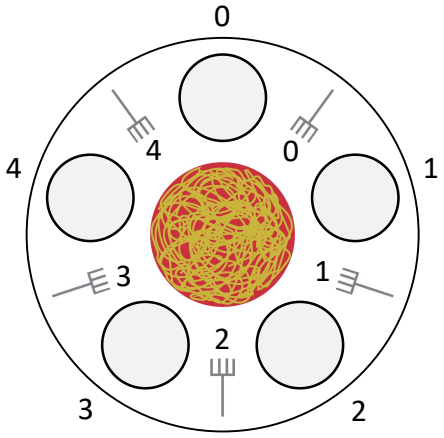
- Each has an assigned seat at the table
- Between each philosopher is a fork

Each independently alternates doing

- Thinking for a while and
- Eating spaghetti for a while
- The supply of spaghetti is unlimited

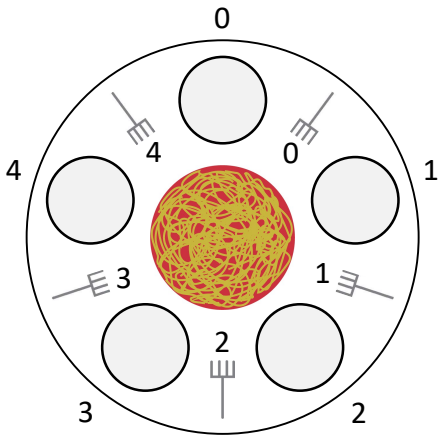
A philosopher needs two forks to eat

- Must be the fork to her left and to her right



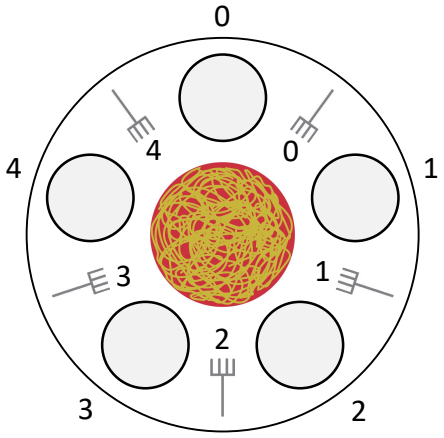
The Life of a Philosopher

```
philosopher(i) {        // i = 0 ... 4
    think . . .
    pick up one fork
    pick up the other fork
    eat . . .
    put down one fork
    put down the other fork
}
```



Specific Rules

- Each philosopher can use only the fork to her left and the fork to her right
- A philosopher can pick up only one fork at a time
- A philosopher must hold both forks simultaneously to eat
- ***It must be possible for two philosophers to eat at the same time***



A Solution to the Dining Philosophers Problem

```
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t  ready[5];

int avail[5] = { 2, 2, 2, 2, 2 };      // count of forks logically available to philosopher i

void pickup_forks(int i) {
        . . .
}

void putdown_forks(int i) {
        . . .
}
```

A Solution to the Dining Philosophers Problem

```
int main() {
   pthread_t  phils[5];
   int args[5];
   int i;

   for (i = 0; i < 5; i++)
      pthread_cond_init(&ready[i], NULL);

   for (i = 0; i < 5; i++) {
      args[i] = i;
      pthread_create(&phils[i], NULL, philosopher, &args[i]);
   }

   for (i = 0; i < 5; i++)
      pthread_join(phils[i], NULL);

   exit(0);
}
```

```
void *
philosopher(void *arg)
{
   int i = *(int *)arg;

   while (1) {
      printf("thinking %d\n", i);
      pickup_forks(i);
      printf("eating %d\n", i);
      putdown_forks(i);
   }

   return NULL;
}
```


The pickup_forks and putdown_forks Procedures

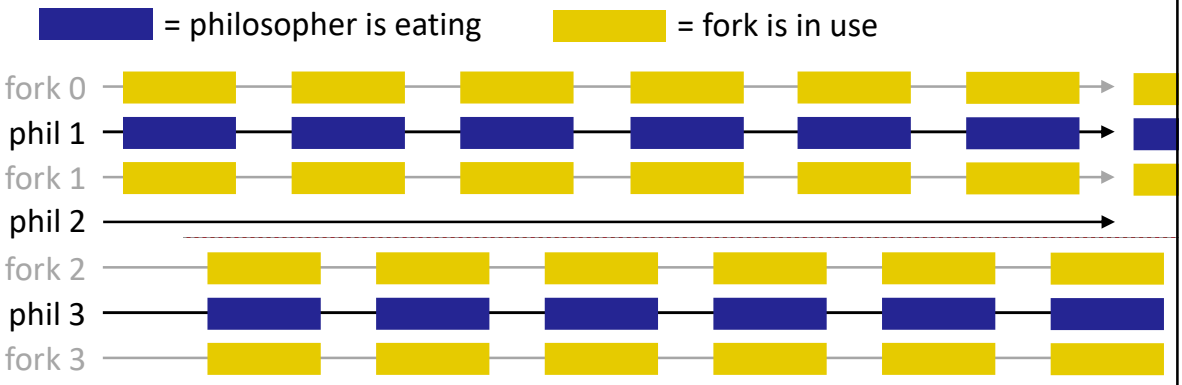
```
void pickup_forks(int i)
{
   pthread_mutex_lock(&mutex);

   while (avail[i] != 2)
      pthread_cond_wait(&ready[i], &mutex);

   avail[(i + 1) % 5]--;      // left
   avail[(i + 4) % 5]--;      // right

   pthread_mutex_unlock(&mutex);
}
```

```
void putdown_forks(int i)
{
   pthread_mutex_lock(&mutex);

   avail[(i + 1) % 5]++;     // left
   avail[(i + 4) % 5]++;     // right

   if (avail[(i + 1) % 5] == 2)
      pthread_cond_signal(&ready[(i + 1) % 5]);
   if (avail[(i + 4) % 5] == 2)
      pthread_cond_signal(&ready[(i + 4) % 5]);

   pthread_mutex_unlock(&mutex);
}
```

This solution works, except that the problem of "starvation" is possible

The Problem of Starvation

Consider the following possible repeating sequence for philosophers 1, 2, and 3



Philosopher 2 can starve if one or both of forks 1 and 2 are always in use

- Solving this is subtle (but not really hard)

How Does errno Work With Threads?

How to keep one thread from changing errno for another thread

- errno is "supposed to be" a global variable, as in

  extern int errno; ⟵ ***do not do this***

- But that would mean a single errno shared by all threads in that process
  - A kernel call by one thread ends up setting errno to indicate some problem
  - Another thread also does some other kernel call which can also set errno
  - The second thread's errno value thus overwrites the errno value that the first thread wanted to be able to see
- Historically, errno's definition was created before Unix supported threads
  - For modern systems, it still seems like a global variable, but it really isn't one

How to Use errno Correctly

Do <u>not</u> declare errno yourself

- Let #include <errno.h> do it correctly for you
- Different systems declare it differently, so don't try to do it yourself
- It will work, e.g., on either the left or the right of an assignment

```
errno = something
something = errno
```

or any other forms of expressions involving errno

If you are curious (you do not need to know this), here's how its done on CLEAR

```
#define errno (*__errno_location())
```

__errno_location() returns a pointer to a thread-specific location for its errno

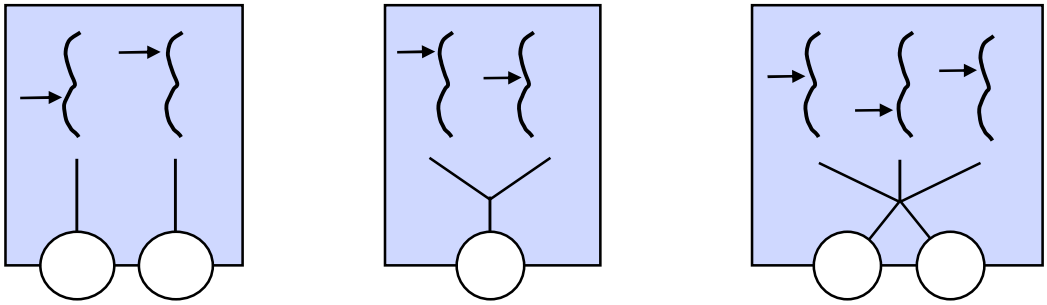
Issues in Thread Scheduling

A thread scheduler has many things to manage and keep track of

- Giving each thread an (approximately) "fair" share of the CPU time
  - Thread context switching to divide the CPU time that the kernel gives you among the user-level threads for that process
- Managing threads that are blocked on I/O
  - Knowing which threads ***should*** get a share of CPU time
- Keeping track of and joining threads when requested
  - Know which threads have completed and not yet been joined
  - Managing a join request for some thread
- Reclaiming thread resources after a threads complete
  - Example: free the thread's stack (it was automatically allocated at create)

Kernel-Level Threads vs. User-Level Threads

Can multiplex user-level threads on kernel-level level threads in many ways



One kernel-level thread per user-level thread

One kernel-level thread for multiple user-level threads

Many user-level threads on many kernel-level threads

Kernel-Level Threads vs. User-Level Threads

Can multiplex user-level threads on kernel-level level threads in many ways

- ***1:1 (kernel-level threading)*** – each user-level thread is mapped to a ***separate*** thread in the kernel
  - Kernel is in control of scheduling each thread, but expensive
- ***N:1 (user-level threading)*** – all user-level threads are mapped to a ***single*** thread in the kernel
  - Kernel doesn't even need to know that threading is being used
  - Scheduling of threads is done at the user level, much less expensive
- ***N:M (hybrid threading)*** – N user-level threads mapped to M kernel threads
  - Flexible dynamic user-level scheduling between user and kernel scheduling
  - Simple example: number of kernel threads (M) = the number of CPU cores

The Effect of a Kernel-Level Thread Blocking

A kernel-level thread blocks when doing something that the kernel blocks it for

- Example: a user-level thread starts some blocking I/O operation
  - The kernel-level thread currently running it is blocked by the kernel
  - So that kernel-level thread no longer available to run any user-level threads
- The other kernel-level threads handling that process's threads can still run
  - But now less CPU resources to run threads in this process
- And if there's only one kernel-level thread for that process's threads
  - Then ***all*** user-level threads in that process are effectively blocked
- Or if now all kernel-level threads for that process are blocked
  - Then again, all user-level threads in that process are effectively blocked

Blocking for I/O in a User-Thread is Very Bad

This kind of blocking can be avoided and <u>needs</u> to be avoided

- Blocking for I/O would block the current kernel-level thread
  - No longer available to service other user-level threads for this process
  - And could (depending on setup) be the last kernel-level thread for it
- ***It is thus important to provide I/O multiplexing and non-blocking I/O within the threads library***
  - To make sure user-level threads don't block on I/O
  - And to enable the thread scheduler to wake up appropriate threads when they have pending I/O that is ready

Be Careful About Signals and Thread Scheduling

As we've seen, a signal can happen anytime, asynchronously

- As always, if you have a signal handler for that signal
  - Be careful what you do in that signal handler
  - Be careful when you should block and can unblock that signal
- This issue can particularly impact thread scheduling
  - A signal may relate to something that directly impacts the scheduler
  - A signal handler may need to read and/or modify some state used by the thread scheduler
  - For example, SIGPROF (sort of similar to SIGALRM) can be used to tell the scheduler to switch to running some other thread now
  - But signal could occur as the scheduler is switching for some other reason, or as some new thread is being created or an existing thread terminated