Week 8

Threads and concurrency

School of Information Technology and Electrical Engineering
The University of Queensland

CSSE 2310 7231

Outline

- Threads
  - Programming with pthreads
- Synchronization

References

Bryant & O'Halloran 13.3

"Programming with POSIX Threads", D. Butenhof, 1997

(Glass & Ables don't talk about threads at all)

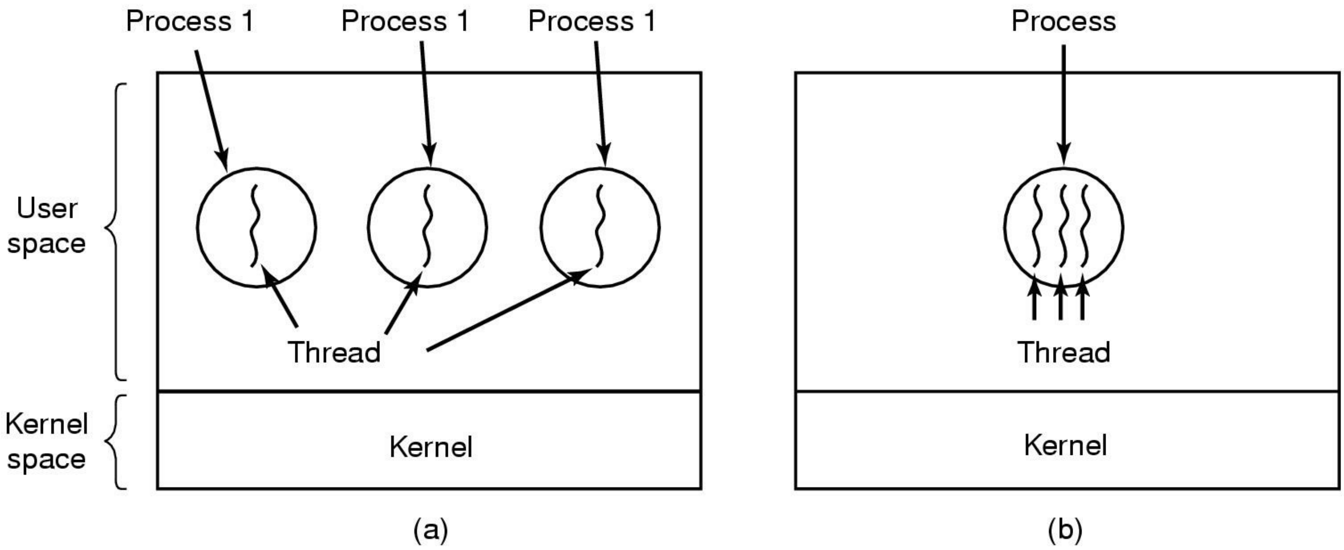
https://computing.llnl.gov/tutorials/pthreads

Threads

A process may have multiple **threads** of control

- Threads share code, data, open files etc but have separate control flows
  - Have to be careful about accessing shared resources!
- Threads have id's, need context switching etc

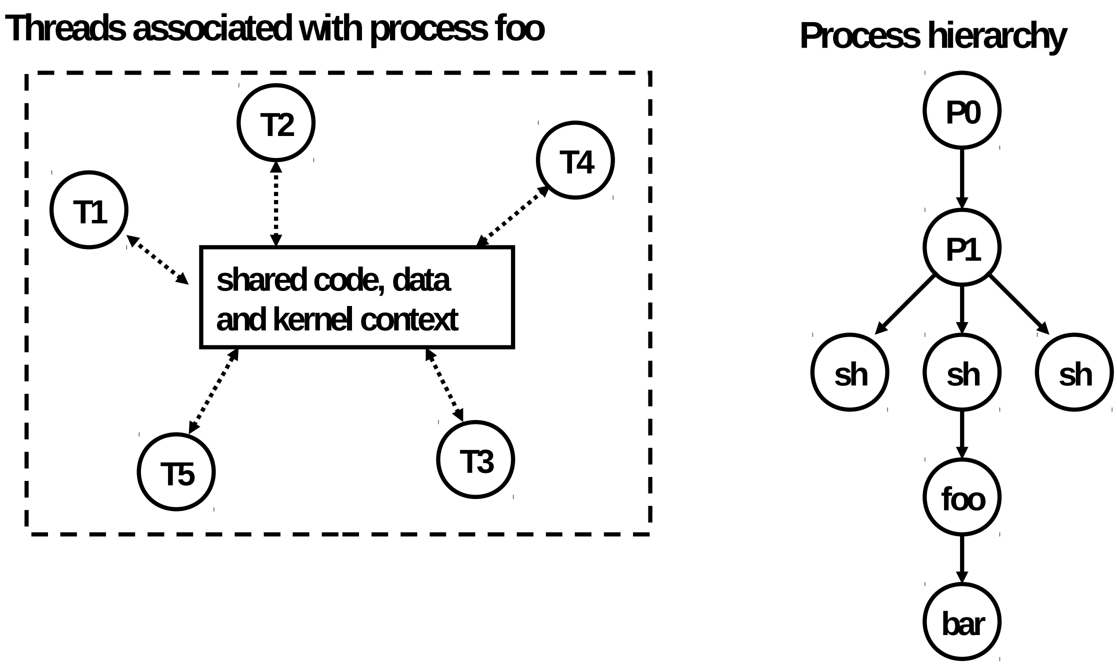
Threads



(a) Three processes each with one thread
(b) One process with three threads

Logical View of Threads

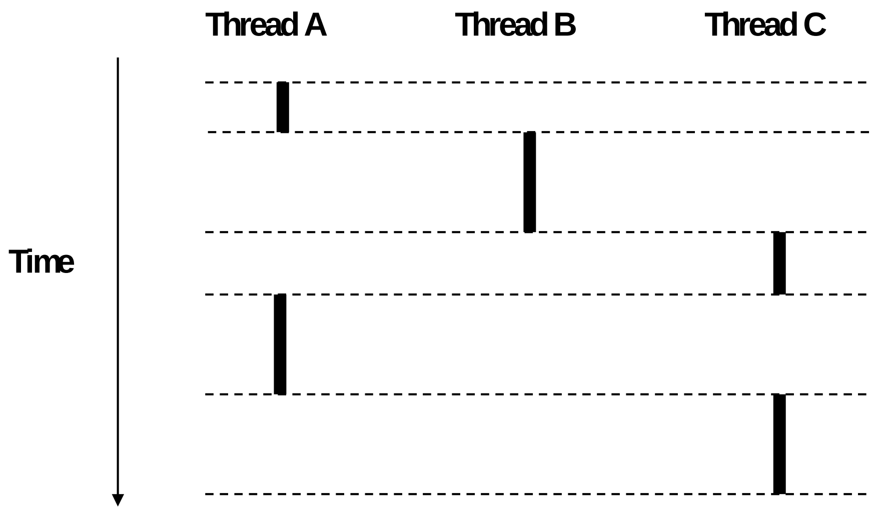
Threads associated with a process form a pool of peers
- Unlike processes which form a tree hierarchy



Concurrent Thread Execution

Two threads run concurrently (are concurrent) if their logical flows overlap in time

Otherwise, they are sequential.



Examples:

- Concurrent
  - A & B, A&C
- Sequential
  - B & C

Threads vs. Processes

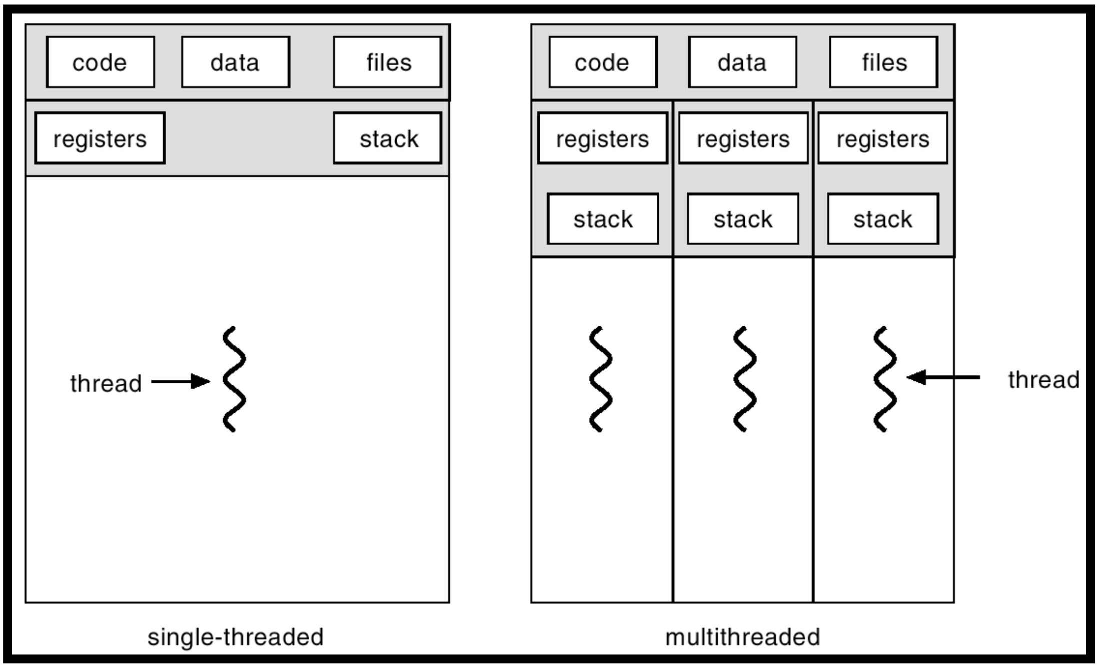
How threads and processes are similar

- Each has its own logical control flow
- Each can run concurrently
- Each is context switched

How threads and processes are different

- Threads share code and data, processes (typically) do not
- Threads are somewhat less expensive than processes
  - Process control (creating and reaping) is twice as expensive as thread control
  - Linux/Pentium III numbers:
    - ~20K cycles to create and reap a process
    - ~10K cycles to create and reap a thread

Single and Multithreaded Processes



Benefits of Threads

Responsiveness
- e.g. one thread for UI, another for computation

Resource Sharing
- Easier to share memory between threads than processes

Economy
- Cheaper to start/switch threads than processes

Utilization of multi-processor (MP) architectures

What about google chrome?

Multithreading Models

Many-to-One (User Threads)

- Threads implemented in user space
  - Packages are available to help with this
- OS knows nothing about them

One-to-One

- Threads implemented in kernel space, one kernel thread per user thread

Many-to-Many

- Hybrid model

Posix Threads (Pthreads) Interface

POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

API specifies behavior of the thread library, implementation is up to development of the library

Common in UNIX operating systems

CSSE 2310 7231

Programming with Pthreads

Thread types

- pthread_t – similar to *pid*
- opaque type

Thread operations

- pthread_create
- pthread_join – similar to *waitpid* (there is no equivalent to *wait*)

CSSE 2310 7231

threads1

pthread_create takes a function pointer to start the thread

pthread_join waits for a specific thread

```
int main() {
   pthread_t tid;
   pthread_create(&tid, NULL, thread1, NULL);
   pthread_join(tid, NULL);
   printf("Hello from first\n");
   exit(0);
}

void *thread1(void *vargp) {
   printf("Hello from second\n");
   return NULL;
}
```

threads2

pthread_self

pthread_exit

```
int main() {
   pthread_t tid;
   pthread_create(&tid, NULL, thread2, NULL);
   printf("Thread %d exiting\n", pthread_self());
   pthread_exit(NULL);
   return 0;
}

void *thread2(void *vargp) {
   printf("Thread %d exiting\n", pthread_self());
   return NULL;
}
```


threads3

pthread_cancel

```
int main() {
   pthread_t tid;
   pthread_create(&tid, NULL, thread3, NULL);
   printf("Killing %d\n", tid);
   pthread_cancel(tid);
   return 0;
}

void *thread3(void *vargp) {
   printf("Thread %d exiting\n",pthread_self());
   return NULL;
}
```

threads4

pthread_detach

```
int main() {
   pthread_t tid;  int status;
   pthread_create(&tid, NULL, thread4, NULL);
   pthread_detach(tid);
   printf("Trying to join %d\n", tid);
   status = pthread_join(tid, NULL);
   if(status)printf("Failed to join thread %d\n",tid);
   pthread_exit(NULL);
   return 0;
}
void *thread4(void *vargp) {
   printf("Thread %d exiting\n",pthread_self());
   return NULL;
}
```


Thread lifecycle

Possible states:

- Ready
- Running
- Blocked
- Terminated
  - Recycling

Compare with process states

Sharing data

What data is shared?

- Global variables – one copy per process
- Local variables – one copy per thread
- Static variables – one copy per process
  - declared multiple times, only one copy exists though!

CSSE 2310 7231

Sharing data

What is output by the following?

```
char **ptr;
int main() {
    int i;
    pthread_t tid;
    char *msgs[N] = {"Hello from foo“,"Hello from bar”};
    ptr = msgs;
    for (i = 0; i < N; i++)
       pthread_create(&tid, NULL, thread5, (void *)i);
    pthread_exit(NULL);
}
void *thread5(void *vargp)
{
  int myid = (int)vargp;
  static int cnt = 0;
  printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
}
```

Sharing data

What does the following output?

```
int count;
int main() {
...
    count = 0;
    /* Create two thread6's, wait for them to finish */
...
    if (count != ITERATIONS * 2)
            printf("Error: %d\n",count);
...}

void *thread6(void *vargp){
    int i;
    for(i = 0; i < ITERATIONS; i++) count++;
    return NULL;
}
```


Sharing data

Race condition

- Global variable count accessed by multiple threads
- The two threads read, then increment, then write back
- Not a single operation

How do we stop this?

Atomic operations

- Must be run without interruption

Sharing data

Race condition

- Global variable count accessed by multiple threads
- The two threads read, then increment, then write back
- Not a single operation

How do we stop this?

Atomic operations

- Must be run without interruption

Critical Section

A critical section of a thread is a segment of code that shouldn't be interleaved with another thread's critical section.

Note that these threads could be in different processes.

Safety and coordination

In the following section, we will address two tasks:

- Protecting critical sections [mutual exclusion]
- Waiting (efficiently) for conditions to be satisfied.

Two (similar) approaches:

- Semaphores
- pthread_mutexes

Semaphores

Threads & processes

Synchronization

- Concurrent access to shared data may result in data inconsistency
  - Remember, concurrent means interleaved threads of control
- Example of problem

```
int inUse; /* shared var, 1 if resource in use */
```

Time

Thread One		Thread Two
`if(!inUse) {`		
	Variable read interleaved between read and set (write)	`if(!inUse) {`
`  inUse=1;`		
		`  inUse=1;`
`  /* use resource */`		`  /* use resource */`
`  ...`		`  ...`
`}`		`}`

Processes?

- This type of problem can occur whenever there are shared resources. For example:
  - Files
  - Shared variables (if shared memory has been mapped).

Semaphores

Special integer variable

- after initialisation, accessed only through two atomic operations
  - atomic = indivisible
    - No interleaving will happen
- Operations on semaphore S
  - ```
    wait(S) {
        while(S <= 0) {
            ; /* do nothing */
        }
        S--;
    }
    ```
  - ```
    signal(S) {
        S++;
    }
    ```


Semaphores (cont.)

`wait(S)` also known as `P(S)`

- based on Dutch word Proberen (to test)

`signal(S)` also known as `V(S)`

- based on Dutch word Verhogen (to increment)

Semaphore value can never be negative

Need hardware/OS support to ensure that operations are indivisible

How to Use Semaphores

Associate a semaphore S, initially 1, with each shared variable (or set of shared variables)

Surround corresponding critical section with wait(S) and signal(S) operations:

```
wait(S)
 ...critical region...
signal(S)
```

This is a binary semaphore - always 0 or 1

Semaphore ensures mutually exclusive access to critical region

- Binary semaphores used for mutual exclusion often called mutexes

Semaphores for Shared Resources

If n resources available, initialise semaphore to n

- allows up to n users

Generalization of mutex

Busy waiting

We wrote:

```
wait(S) {
    while(S <= 0) {
        ; /* do nothing */
    }
    S--;
}
```

but OS doesn't actually busy wait

- Process shifted to waiting queue
- Process shifted to ready queue when semaphore available
    - If more than one process waiting on a particular semaphore, need to choose process appropriately to prevent starvation (i.e. one process waiting indefinitely)

Deadlock

Some processes may wait forever, e.g.

Process One		**Process Two**
`wait(S1);`		`wait(S2);`
`wait(S2);`	May get stuck here ↔	`wait(S1);`
…		…
`signal(S1);`		`signal(S2);`
`signal(S2);`		`signal(S1);`

Need deadlock avoidance strategies

- Beyond scope of this course

Semaphore APIs

System V Semaphore API
- Very complicated to use
  - `semget()`, `semctl()`, `semop()`

POSIX Semaphore API
- “unnamed/memory” semaphores.
- named semaphores.

POSIX memory semaphores

Only work where all threads/processes can see the memory the semaphore uses. ie threads in one process or processes with shared memory.

Disappear when process dies.

```
sem_t mine;

sem_init(&mine, 0, initval);
 …
sem_wait(&mine);
/* critical section */
sem_post(&mine);
…
sem_destroy(&mine);
```

CSSE 2310 7231

POSIX named semaphores

Identified by name. Processes do not need to share memory.

Persist until they are explicitly removed or the system is rebooted.

```
sem_t* mine=sem_open("/jsempre", O_CREAT, initval);
…
sem_wait(&mine);
…
sem_post(&mine);
…
sem_unlink("/jsempre");
```


CSSE 2310 7231

Semaphore APIs

POSIX Semaphore API

- `sem_wait()   /* wait() or P() */`
- `sem_post()   /* signal() or V() */`
- Other functions also, e.g.
  - `sem_getvalue()` – return value of semaphore
  - `sem_trywait()` – don't block if semaphore is 0
  - `sem_timedwait()` – wait, but only for a while

pthread_mutex

- threads

Invariants, critical sections and predicates

Invariants: assumptions about the relationsip between variables

– eg. state of queue

Critical section: code that affects shared state

– eg. removing data from queue

Predicate: logical expression to describe invariant

– eg. “queue is empty”

Sharing data revisited

What does the following output?

```
int count;
int main() {
...
    count = 0;
    /* Create two thread6's, wait for them to finish */
...
    if (count != ITERATIONS * 2)
            printf("Error: %d\n",count);
...}

void *thread6(void *vargp){
    int i;
    for(i = 0; i < ITERATIONS; i++) count++;
    return NULL;
}
```


Mutex

The idea:

- mutex allows only one thread to access a resource
- other threads block until the mutex is released

pthread_mutex_t

pthread_mutex_init

pthread_mutex_lock

pthread_mutex_unlock

Sharing data with mutexes

```
int count;
int main() {
...
    count = 0;
    /* Create two thread6's, wait for them to finish */
...
    if (count != ITERATIONS * 2)
            printf("Error: %d\n",count);
...}
void *thread7(void *vargp){
   int i;
   for(i = 0; i < ITERATIONS; i++){
        pthread_mutex_lock(&mutex);
        count++;
        pthread_mutex_unlock(&mutex);
   }
   return;
}
```


More mutexes

Don't always want to block

– pthread_mutex_trylock

How big should a mutex be?

– One mutex per variable?

– One mutex for many variables?

What happens in the following code?

```
int threadA (void *vargp) {
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
       ... Do some stuff ...
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}
int threadB (void *vargp) {
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(&mutex1);
       ... Do some stuff ...
    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);
    return NULL;
}
```

CSSE 2310 7231

Condition variables

Mutexes make sure only one thread can access data at a time

What if we want a thread to wait until a variable reaches a certain value?

Polling

Condition variables

- send signal to threads waiting
- used in conjunction with a mutex

Condition variables

An example:

- two threads, one writing to a queue, the other reading from it
- In order to access the queue, both need to lock a mutex
- Once locked, the reader discovers the queue is empty
- Reader waits on a condition variable (which unlocks the mutex)
- The writer locks the mutex, accesses the queue, adds an item, unlocks mutex
- The reader’s wait returns, with the mutex locked again, allowing it to access the queue

Condition variables

pthread_cond_t

pthread_cond_init

pthread_cond_destroy

pthread_cond_wait

pthread_cond_timedwait

pthread_cond_signal

pthread_cond_broadcast

Condition variables

wait always returns with the associated mutex locked

use for signalling, NOT mutual exclusion – that’s what mutexes are for!

condition variable should be associated with only one predicate

Using condition variables

cond.c

Notes:

- Spurious wakeups are possible – need to check predicate again!
- Check predicate!
- Check return values!

Attributes

of threads

– pthread_attr_init

of mutexes

– pthread_mutexattr_setprotocol

of condition variables

– pthread_condattr_init

Issues

sleep?

exec?

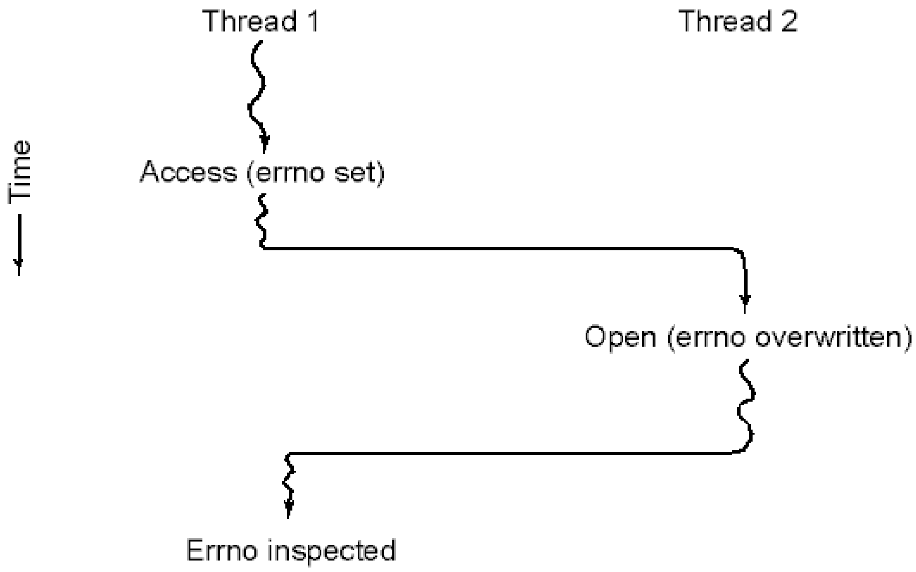
fork?

Signals?

Shared libraries?

Making Single-Threaded Code Multithreaded

Conflicts between threads over the use of a global variable (e.g. errno)



Thread Safety

Functions called from a thread must be **thread-safe**

Beware
- Shared variables
- Static variables in functions
- Relying on persistent state between invocations
- Calling thread-unsafe functions

Examples:
- `pread()` instead of `read()`
- `localtime_r()` instead of `localtime()`

Summary

Threads
- Creating
- Synchronizing using mutexes
- Communicating using condition variables

Programming with threads