Threads

Network Programming

Servers

- Typically offer some resource/service
  - Potentially, multiple users may want to access this simultaneously e.g., web pages, email, etc.
- Both processes and threads offer the ability for multiple clients
  - Different paradigms, difference performance, different problems

Processes

- Created via **`fork()`**
  - Fairly heavy-weight
  - Most platforms use COW optimization
  - Programs often **`exec()`** afterward

- Have to use IPC after the **`fork()`**
  - Communication is difficult
  - Typically use **`mmap()`** or shared memory
  - See the **`pipe()`** system call as well

fib_fork.c

```c
int main()
{
    int children[NUM_CHILD];

    for (int i = 1; i < NUM_CHILD; i++) {
        int cid = fork();

        if (cid == 0) {
            printf("Child %d is %d\n", i, fib(i));
            return 0;
        }
        else {
            children[i] = cid;
        }
    }

    for (int i = 0; i < NUM_CHILD; i++) {
        waitpid(children[i], 0, 0);
    }

    return 0;
}
```

Rebuild Environment

- COW takes care of this
  - Identical copy means everything can be shared
  - When one process changes something shared between multiple processes, a copy is made
  - Page tables are included

Process Communication

- Signals
  - kill sends signals to your process
- Pipes
  - **`mkfifo()`**
- Shared memory
  - **`mmap()`**
- Sockets

Threads

- Shared address space of a single process
- Sometimes called lightweight processes
- Mitigates some of the problems with processes e.g., expensive creation, communication difficulties
  - Introduces new problems of course… Race conditions!

How lightweight?

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags.

Improved Performance

- Threads enjoy better caching effects
- Possible to exploit higher performance due to single namespace
- Reduced overhead due to fewer copies (typically not necessary) – same address space!

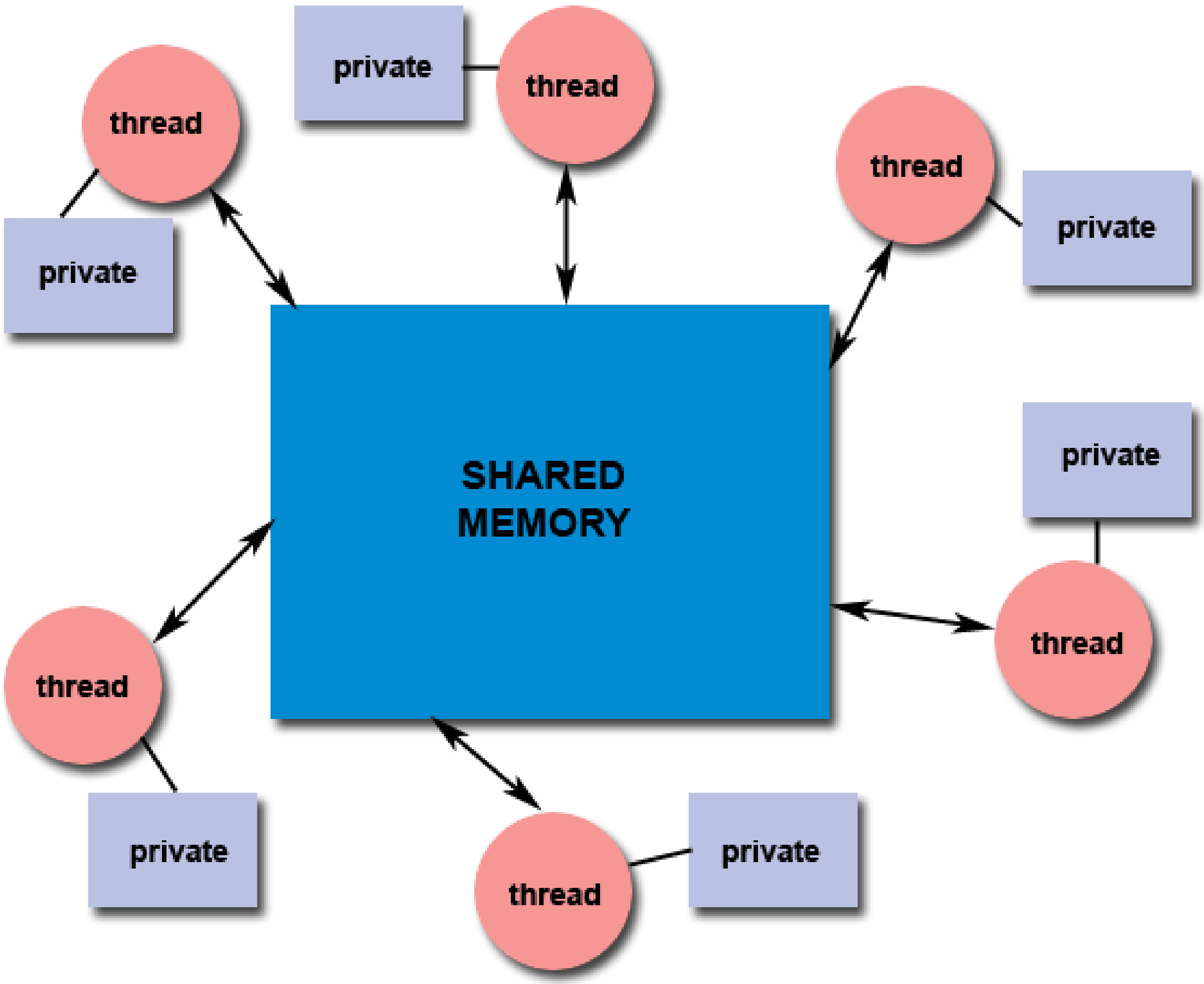
Better Utilization

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Are threads a good choice?

- Many issues to consider:
  - Partitioning
  - Communications
  - Synchronization
  - Program complexity
- Does the program patterns lend themselves well to threads?
  - Independent, concurrent tasks?

Shared Memory



Great!

- Fast access to anything/everything!
- Wait a minute…
- Financial perspective:
  - $balance += $200
  - Is this a thread-safe operation?

Bank Balance

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

- Well that's not good!
- We need to deal with that, too

pthreads

- **`pthread_create()`**
  - Create a new thread with default parameters
- **`pthread_join()`**
  - Wait for a thread to complete
  - Similar to **`waitpid()`**
- **`pthread_attr_init()`**
  - Create thread attribute object

Thread Communication

- Everything is shared!
  - Except registers, stack, and thread-local storage (TLS)
  - TLS-code is a maintenance nightmare; platforms have wildly varying levels of support

- Shared state updates have to be manually synchronized
  - Locks, mutexes, thread-safe code, etc.

fib_thread.c

```c
int main()
{
    pthread_t children[NUM_CHILD];

    for (long i = 1; i < NUM_CHILD; i++) {
        pthread_t tid;
        int val = pthread_create(&tid, NULL, fib, (void*)i);

        if (val < 0) {
            return -1;
        }
        else {
            children[i] = tid;
        }
    }

    for (int i = 1; i < NUM_CHILD; i++) {
        int *ret_val;
        pthread_join(children[i], (void**)&ret_val);
        printf("Child %d is %d\n", i, (int)ret_val);
    }

    return 0;
}
```

fib_thread.c pt. 2

```c
void * fib(void * v)
{
    long n = (long)v;
    if (n == 0)
        return 0;
    if (n == 1)
        return (void *)1;
    if (n == 2)
        return (void *)1;

    void *ret1 = fib((void*)(n-1));
    void *ret2 = fib((void*)(n-2));
    long ret3 = (long)ret1 + (long)ret2;
    return (void*)ret3;
}
```

Thread Termination (1/2)

- When does a pthread_create() terminate?
  - From the man page…
- It calls pthread_exit(3), specifying an exit status value that is available to another thread in the same process that calls pthread_join(3).
- It returns from start_routine(). This is equivalent to calling pthread_exit(3) with the value supplied in the return statement.

Thread Termination (2/2)

- It is canceled (see pthread_cancel(3)).
- Any of the threads in the process calls exit(3), or the main thread performs a return from main(). This causes the termination of all threads in the process.

Thread Coordination

- By default, all threads are *joinable*
  - Similar to children processes requiring **`waitpid()`** calls
- You can call **`pthread_detach()`** to release it from its need to be joined
- We used joining in our previous example!

Bank Balance Pt. 2

- We need *mutual exclusion* (mutex) while accessing *shared* data/variables

- We can grab our mutex variable
  - If we have it, no one else does!
  - This means our data access is protected (locked)
  - We must remember to unlock the mutex!

Documentation

- If you try to do "man pthread_join" it probably worked
- If you try to do "man pthread_mutex_init" (introduced on the next slide), you're probably not so lucky
  - In Ubuntu/WSL,
    sudo apt install manpages-posix-dev
    will fix this

Using a mutex

- **`pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`**
  - Or dynamically with **`pthread_mutex_init()`**

- **`pthread_mutex_lock (mutex)`**
- **`pthread_mutex_trylock (mutex)`**
- **`pthread_mutex_unlock (mutex)`**

Using Condition Variables

- `pthread_cond_t mycondv = PTHREAD_COND_INITIALIZER;`
  - Or dynamically with `pthread_cond_init()`

- `pthread_cond_wait(cond, mtx)`
- `pthread_cond_signal(cond)`

produce()

```c
void produce(struct buf_t *bptr, int val)
{
    Pthread_mutex_lock(&bptr->b_mutex);
        /* Wait if buffer is full */
    while (bptr->b_nitems >= BUFFSIZE)
        Pthread_cond_wait(&bptr->b_cond_producer, &bptr->b_mutex);

        /* There is room, store the new value */
    printf("produce %d\n", val);
    bptr->b_buf[bptr->b_nextput] = val;
    if (++bptr->b_nextput >= BUFFSIZE)
        bptr->b_nextput = 0;
    bptr->b_nitems++;

        /* Signal consumer */
    Pthread_cond_signal(&bptr->b_cond_consumer);
    Pthread_mutex_unlock(&bptr->b_mutex);
}
```

consume()

```
int consume(struct buf_t *bptr)
{
    int     val;

    Pthread_mutex_lock(&bptr->b_mutex);
        /* Wait if buffer is empty */
    while (bptr->b_nitems <= 0)
        Pthread_cond_wait(&bptr->b_cond_consumer, &bptr->b_mutex);

        /* There is data, fetch the value */
    val = bptr->b_buf[bptr->b_nextget];
    printf("consume %d\n", val);
    if (++bptr->b_nextget >= BUFFSIZE)
        bptr->b_nextget = 0;
    bptr->b_nitems--;

        /* Signal producer; it might be waiting for space to store */
    Pthread_cond_signal(&bptr->b_cond_producer);
    Pthread_mutex_unlock(&bptr->b_mutex);

    return(val);
}
```

Rusty Quote

- “Deadlocks are problematic, but not as bad as data corruption. Code which grabs a read lock, searches a list, fails to find what it wants, drops the read lock, grabs a write lock and inserts the object has a race condition. If you don't see why, please stay the fuck away from my code.”

- https://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/x441.html

Threads Cont.

- We may revisit some of this material in a later lecture by covering Ch 26 in your textbook
  - You may still find it a helpful read if you really didn't get threads, and the man pages are scary
- Not an OS course so don't want to get too deep into it
- Generally easier to avoid threads, but you may see them in real apps. They can be powerful, but a pain to work with.

Lab

- Lab4.pdf
- Simple threading lab
  - More than 1 argument needed to add()
  - Shouldn't need to use globals (except NUM_CHILD)
  - Remember, all threads share memory