Threads
Refresher: Processes and threads

- Processes are expensive to create.
- It takes quite a bit of time to switch between processes.
- Communication between processes must be done through an external structure – files, pipes, shared memory.
- Synchronizing between processes is cumbersome.
- *Is there another model that will solve these problems?*
Thread model

• A thread is a single control flow through a program
  – What is a “control flow”?  
  – How is control flow represented?

• A program with multiple control flows is multithreaded
General idea

- Program runs single-threaded (one flow)
- Create a bunch of threads (multiple flows)
- Wait for all thread control flows to finish (Join all threads).
- Continue as single flow
Advantages

• Communication between threads is cheap
  – they can share variables!
• Threads are “lightweight”
  – faster to create
  – faster to switch between
### Performance: processes vs threads

#### Time results:

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

Source (Lawrence Livermore National Labs): [https://computing.llnl.gov/tutorials/pthreads/](https://computing.llnl.gov/tutorials/pthreads/)
Today: Pthreads

- POSIX threads (pthreads) is the most commonly used thread package on Unix/Linux
General idea

Create N threads:
pthread_create(..)

Wait for all to finish:
pthread_join(..)
pthread_create

```c
int pthread_create(pthread_t *tid,
                  pthread_attr_t *attr,
                  void *(*func)(void*), void *arg);
```

- **tid** uniquely identifies a thread within a process and is returned by the function
- **attr** sets attributes such as priority, initial stack size
  - can be specified as NULL to get defaults
- **func** - the function to call to start the thread
  - accepts one void* argument, returns void*
- **arg** is the argument to func
- returns 0 if successful, a positive error code if not
- does not set `errno` but returns compatible error codes
- can use `strerror()` to print error messages
**pthread_join**

```c
int pthread_join(pthread_t tid, void **status)
```

- **tid** - the tid of the thread to wait for
  - cannot wait for any thread (as in `wait()`)
- **status**, if not NULL returns the `void *` returned by the thread when it terminates.
- a thread can terminate by
  - returning from `func`
  - the `main()` function exiting
  - `pthread_exit()`
More functions

- **void pthread_exit(void *status)**
  - a second way to exit, returns `status` explicitly
  - `status` must not point to an object local to the thread, as these disappear when the thread terminates.

- **int pthread_detach(pthread_tid);**
  - if a thread is detached its termination cannot be tracked with `pthread_join()`
  - it becomes a daemon thread

- **pthread_t pthread_self(void)**
  - returns the thread ID of the thread which called it
  - often see `pthread_detach(pthread_self())`
Passing Arguments to Threads

```c
pthread_t thread_ID;  int fd, result;
fd = open("afile", "r");
result = pthread_create(&thread_ID, NULL,
                        myThreadFcn, (void *)&fd);
if(result != 0)
    printf("Error: %s\n", strerror(result));
```

- We can pass any variable to our thread function.
- It assumes the thread function knows what type it is.
- This example is bad if the main thread alters fd later.
- Also, how do we pass a whole lot more than just an int, a string, etc.? Ideas?
Solution

- “Bundle” in a struct all the data to be passed to a thread
- Use malloc() to create memory for the variable
  - initialize variable’s value
  - pass pointer to new memory via pthread_create()
  - thread function releases memory when done.
- Example:

```c
typedef struct thread_data{
    int fd;
    char name[25];
} tdata;

int result;
pthread_t thread_ID;
```
Example (cont’d)

tdata *p = (tdata *)malloc(sizeof(tdata));
p->fd = fd;  /* assumes fd is defined */
strncpy(p->name, "CSC209", 7);
result = pthread_create(&threadID, NULL,
    myThreadFcn, (void *)p);

void *myThreadFcn(void *p) {
    tdata *mydata = (tdata *) p;
    write(mydata->fd, mydata->name, 7);
    close(mydata->fd);
    free(mydata);
    return NULL;
}
Thread-safe functions

- Not all functions can be called from threads
  - many use global/static variables
  - new versions of UNIX have thread-safe replacements like strtok_r() (re-entrant version)

- Safe:
  - ctime_r(), gmtime_r(), localtime_r(), rand_r(), strtok_r()

- Not Safe:
  - ctime(), gmtime(), localtime(), rand(), strtok(), gethostxxxx()

- Could use synchronization mechanisms (semaphores, mutexes) to protect access but will generally result in poor performance.
Protecting shared resources

• Programs that manage shared resources must protect the integrity of the shared resources
• Operations that modify the shared resource are called critical sections
• Critical section must be executed in a mutually exclusive manner
• Mutexes/Locks or Semaphores are commonly used to protect critical sections
Protecting critical sections

• Code that modifies shared data usually has the following parts:
  – **Entry section**: The code that requests permission to modify the shared data
  – **Critical Section**: The code that modifies the shared variable
  – **Exit Section**: The code that releases access to the shared data
  – **Remainder**: The remaining code
Mutexes/Locks

- **Acquire(m) or Lock(m)**
  - block until the lock is released
  - only the process that acquired the mutex is allowed to run in the critical section

- **Release(m) or Unlock(m)**
  - signals other processes that the mutex is up for grabs
  - only one of the processes waiting on the mutex gets to acquire it
Pthread Mutexes ("Locks")

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

int pthread_mutex_lock(pthread_mutex_t *mp);
int pthread_mutex_trylock(pthread_mutex_t *mp);
int pthread_mutex_unlock(pthread_mutex_t *mp);
int pthread_mutex_destroy(pthread_mutex_t *mp);

• only the thread that locks a mutex can unlock it
• mutexes often declared as globals
Example

pthread_mutex_t myMutex; // Global

int status;
status = pthread_mutex_init(&myMutex, NULL);
if(status != 0)
    printf("Error: %s \n", strerror(status));

// Code snippet executed by all threads
pthread_mutex_lock(&myMutex);
/* critical section here */
pthread_mutex_unlock(&myMutex);

status = pthread_mutex_destroy(&myMutex);
if(status != 0)
    printf("Error: %s\n", strerror(status));
Synchronize shared resources

```c
int var = 0;

void *myThreadFcn(void *p) {
    int *tid = (int *) p;
    var++;
    printf("Thread %d says var=%d\n", i, var);
    return NULL;
}

int main() {
    pthread_t tid[10];
    for (i=0; i<10; i++)
        pthread_create(&t[i], NULL, myThreadFcn, (void *)&i);
    ...
}
```

What’s the problem and how to fix it?
Concurrency bugs

- Hard to detect and reproduce
- Scheduling policy is important (preemptive, non-preemptive)