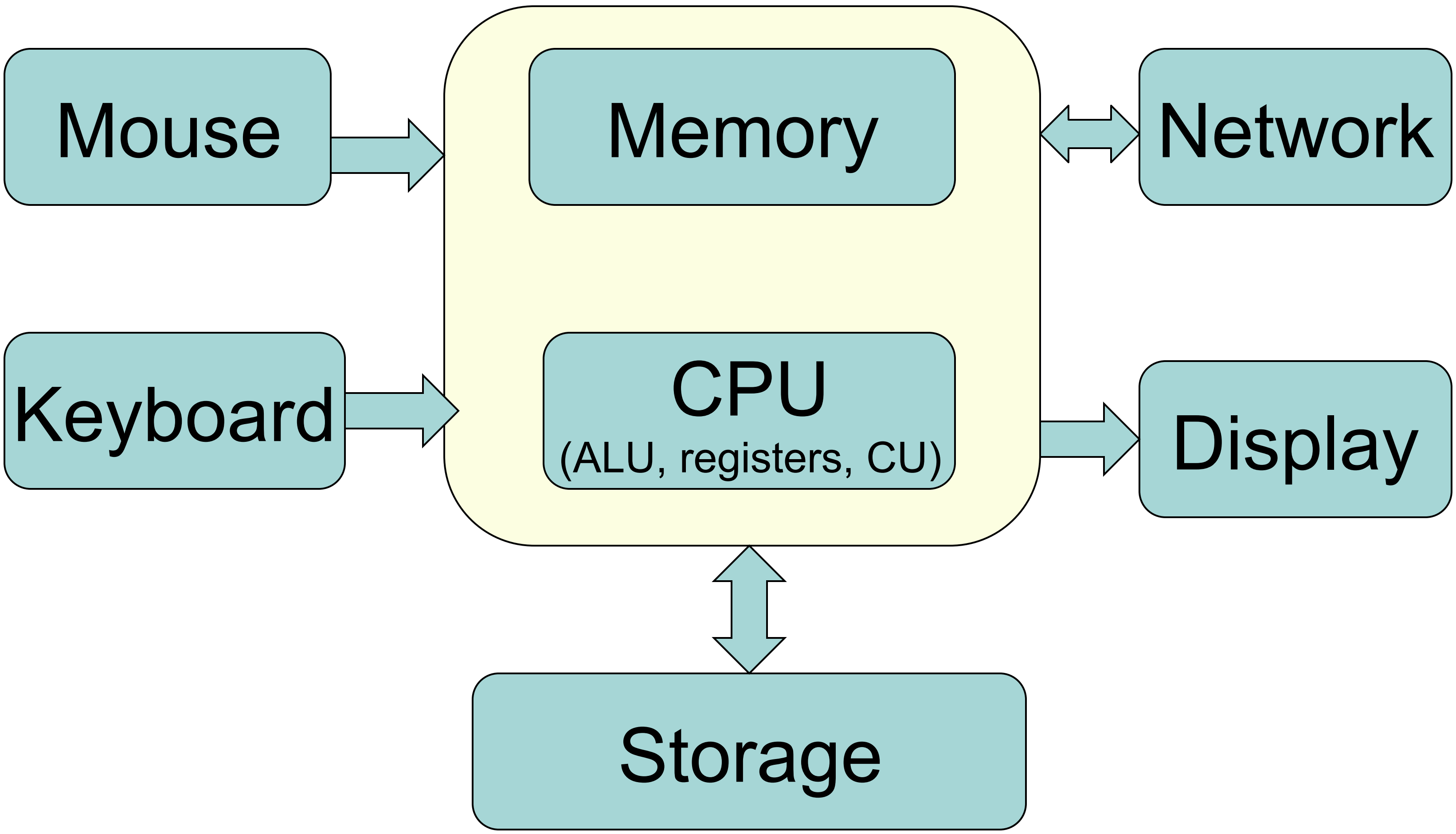
CS240: Programming in C

Lecture 18: Threads

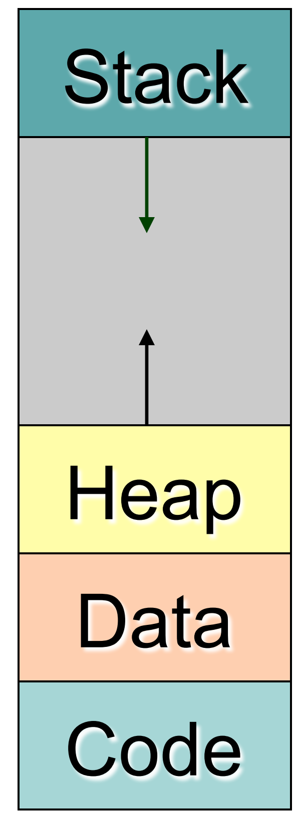
Basic Computer Architecture



Process

- Each process has its own resources and memory
- Resources:
  - Registers
  - Stack, heap, shared libraries, program instructions
  - File descriptors

Virtual Memory



Threads vs Processes

- Process:
  - an independent unit of execution isolated from all other processes and shares no resources
  - can be created by other process (fork, exec)
- Thread:
  - an independent unit of execution that shares resources with other threads
  - exists within a process, but has independent control flow
  - scheduled by the operating system
  - functions to work with threads – different standards, e.g. POSIX

Threads

- **Share common process resources** (like heap and file descriptors)
  - changes made by one thread visible to others
  - pointers have meaning across threads
  - <u>two threads can concurrently read and write to the same memory location</u>
- Maintain their own stack pointer and registers
- Pending and blocked signals

Why Threads?

- Concurrency
  - Expression of a task in the form of multiple, possibly interacting subtasks, that may potentially be executed at the same time.
  - It says nothing about how the subtasks are actually executed.
  - Concurrent tasks may be executed serially or in parallel depending upon the underlying physical resources available.

Concurrency and Parallelism

- Concurrency is concerned with the management of logically simultaneous activities
  - best-fit job scheduling
  - event handling (GUI)
  - web server request
- Parallelism is concerned with performance of concurrent activities
  - weather forecasting
  - simulations

Parallelism

- Parallelism:
  - Execution of concurrent tasks on platforms capable of executing more than one task at a time is referred to as “parallelism”
- Parallelism integrates elements of execution -- and associated overheads
- We typically examine the correctness of concurrent programs and performance of parallel programs.

Why Parallelism

- Resources of a computer (processor, the data-path, the memory subsystem, the disk, and the network) represent bottlenecks.
- Parallelism alleviates all of these bottlenecks.

Parallelism Benefits for Memory

- Most programs are memory bound – i.e., they operate at a small fraction of peak CPU performance (10 – 20%)
- They are, for the most part, waiting for data to come from the memory.
- Parallelism provides multiple pathways to memory – effectively scaling memory throughput as well!

Parallelism Benefits for IO

- I/O (disks) represent major bottlenecks in terms of their bandwidth and latency
- Parallelism enables extraction of data from multiple disks at the same time, effectively scaling the throughput of the I/O
- Example: large server farms (several thousand computers) that ISPs maintain for serving content (html, movies, music, mail)

Parallelism Benefits for CPU

- The process itself is the most obvious bottleneck.
- Processors increasingly pack multiple cores

Challenges

- Coordination
- Synchronization

- Safety and liveness
  - Safety: consistency, nothing bad happens
  - Liveness: progress, something good happens

Multi-threaded Architectures

Shared Memory Model:

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data

Thread-safeness

- Thread-safeness: application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.
- Example:
  - An application creates several threads, each of which makes a call to the same library routine:
  - This library routine accesses/modifies a global structure or location in memory.
  - It is possible that the threads may try to modify this global structure/memory location at the same time.
  - If the routine does not use synchronization constructs to prevent data corruption, then it is not thread-safe.

PThreads and Portability

- POSIX Threads, for short **Pthreads**, is a POSIX standard for threads, defining an API for creating and manipulating threads.
- Although Pthreads API is a standard, implementations can, and usually do, vary
  - a program that runs fine on one platform, may fail or produce wrong results on another platform.
- Example, the maximum number of threads permitted, and the default thread stack size are two important limits to consider when designing a program.

Pthreads API

- Thread management - creating, joining threads etc.
- Mutexes
- Condition variables
- Synchronization between threads using read-write locks and barriers

- Must include **`pthread.h`** header and link with **`pthread`** library

pthread_create

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

- On success, **pthread_create**() returns 0; on error, it returns an error number, and the contents of **thread* are undefined.

pthread_create

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

- `*thread` will be set to contain the id of the new thread.
- this id will be passed to other pthreads functions that require a pthread identifier

pthread_create

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

- **`attr`** structure whose contents are used at thread creation time to determine attributes for the new thread; initialized using **`pthread_attr_init`**. If **`attr`** is NULL, then the thread is created with default attributes.

pthread_create

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

- **`start_routine`** is the function invoked when the thread starts, it's what the thread does.
- **`arg`** is the arguments passed to **`start_routine`**, it can be NULL

pthread_exit

```
#include <pthread.h>
void pthread_exit(void *retval);
```

- This function always succeeds.
- To allow other threads to continue execution, the main thread should terminate by calling **`pthread_exit()`** and not **`exit`**

Example

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define N 5

void* hello(void *id) {
    printf("Hello %ld\n", (long)id);
    pthread_exit(NULL);
}
int main (int argc, char *argv[]) {
    pthread_t threads[N];
    for(long t=0; t<N; t++){
      int rc = pthread_create(&threads[t], NULL,
                                            hello, (void *)t);
      if (rc) exit(-1);
    }
    pthread_exit(NULL);
}
```

Multiple arguments

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};
struct thread_data thread_data_array[NUM_THREADS];
void *PrintHello(void *threadarg){
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}
int main (int argc, char *argv[]) {
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   err = pthread_create(&threads[t], NULL, PrintHello, (void *) &thread_data_array[t]);
   ...
}
```

Joining and Detaching Threads

- **`pthread_join()`** blocks the calling thread until the specified thread id terminates
- A joining thread can match one **`pthread_join()`** call
- A thread created as detached can never be joined
- Use the **`attr`** argument in a **`pthread_create()`** call to set joinable or detachable attributes

Pthread_join

```
#include <pthread.h>
int pthread_join(pthread_t thread, void * retval);
```

- Waits for the thread identified by id **`thread`** to finish. That thread must be joinable.
- If retval is not NULL, then the result from pthread_exit is returned there.
- If multiple thread try to join the same thread the result in undefined.
- On success returns 0, on error a negative number.

Example

```
#include <pthread.h>
...
#define NUM_THREADS! 4
void *BusyWork(void *t) { ... pthread_exit((void*) t);   }
int main (int argc, char *argv[]) {
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   ...
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      err = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      ...         }
   }
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      err = pthread_join(thread[t], &status);
      ...
      printf("Main: completed join with thread %ld having a status
            of %ld\n",t,(long)status);
   }
   printf("Main: program completed. Exiting.\n");
   pthread_exit(NULL);
}
```

Mutual Exclusion

- At most one thread can “acquire” a mutex at any given time.
  - Once the acquiring thread “releases” the mutex, another thread waiting for it can acquire it
- Threads waiting for a mutex are blocked from performing any other work
- Logical errors that can occur when mutexes are used incorrectly
  - Not used when they should be
  - Used when they shouldn’t be

Mutexes

- Protect access to shared data
- Methodology
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - One succeeds
  - Owner manipulates data protected by mutex
  - Owner unlocks
  - Another thread acquires the mutex, and repeats
  - Destroy the mutex

Challenges using mutexes

- Make sure data is consistently protected by the same set of mutexes
- Make sure mutexes properly released
- Ensure deadlock-freedom
- Ensure progress (liveness)