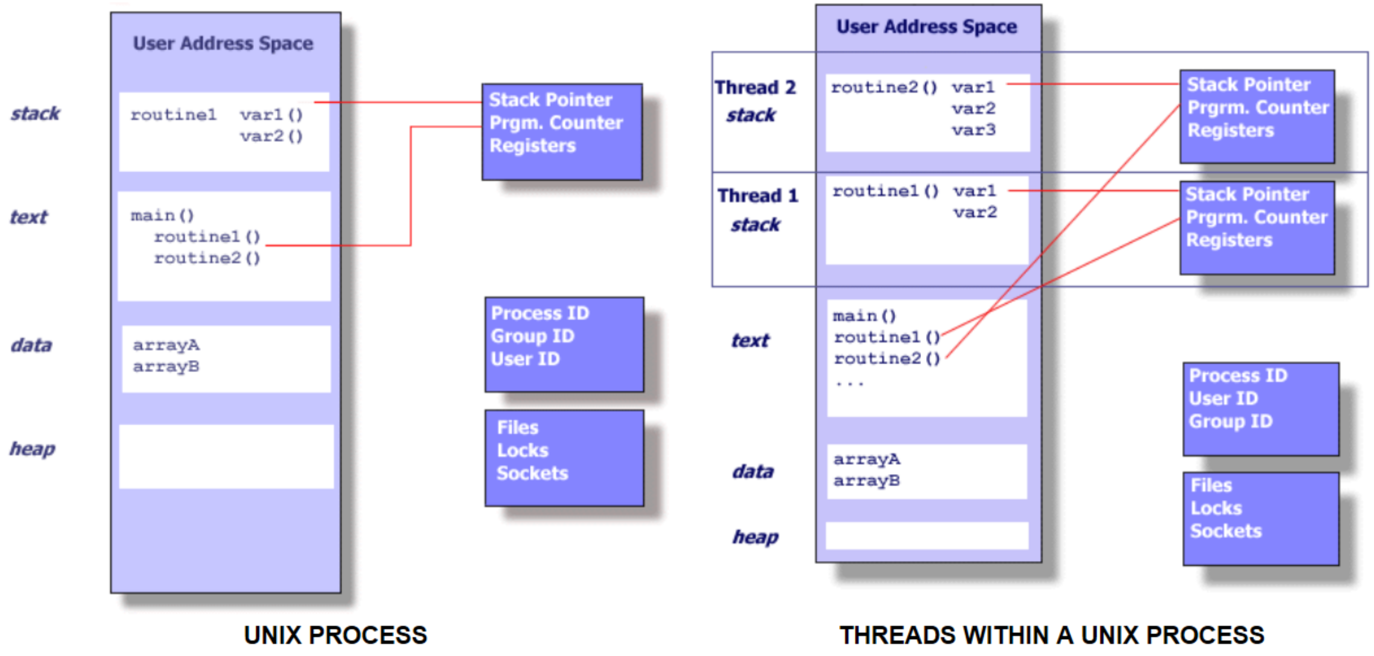
CS214-system programming

- Section 03/08 recitation 8

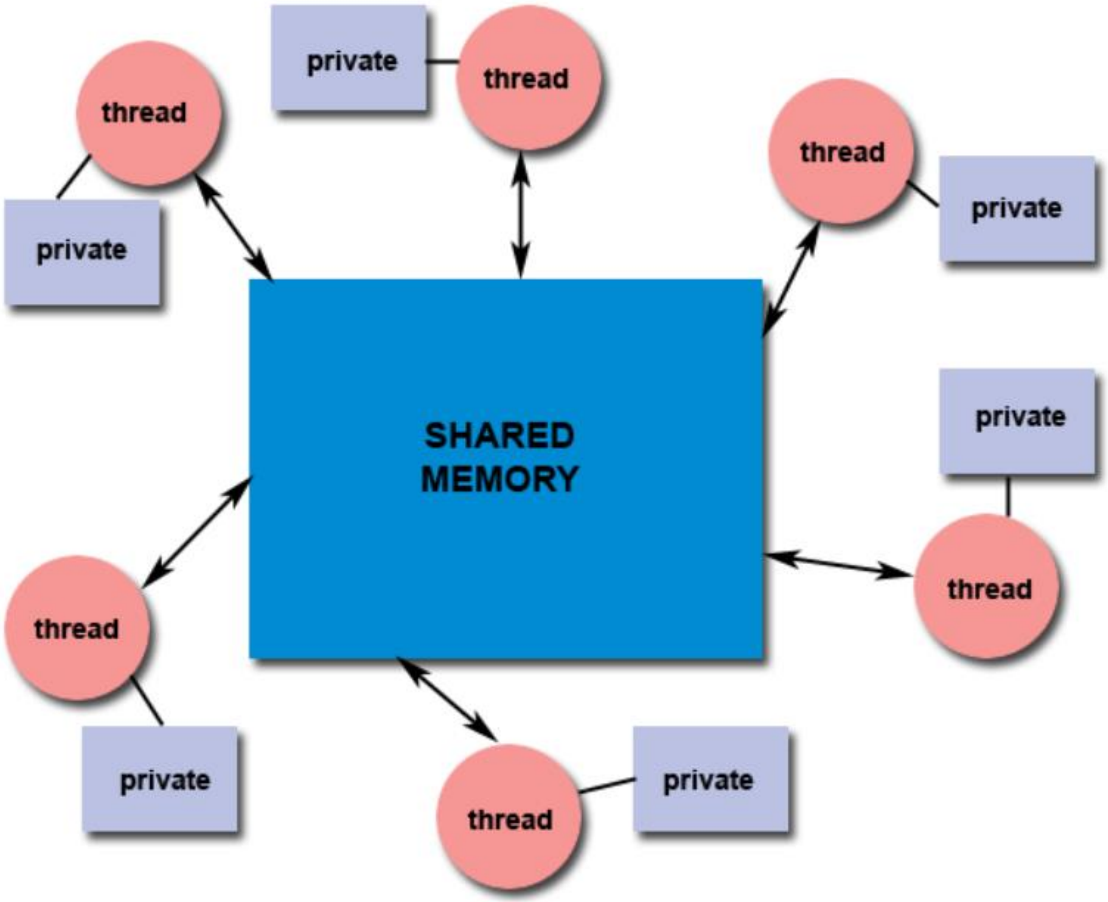
Yunhe Gao
yg397@scarletmail.rutgers.edu

Review: Thread

- A thread is a flow of execution through the process code
- As a process is created, it has at least one running thread
  - Each process can have multiple threads
- A thread can be regarded as a light-weighted process: most of the overhead has already been accomplished through the creation of the corresponding process
- Each has its own program counter, system registers and a stack
  - local variables aren't shared between threads
- User address space and process control block is shared among all the threads for the same process
  - Threads of the same processes share code, global variables and heap
  - Memory allocated from the heap or global variables can be used for sharing data between threads



Threads: Shared Memory Model



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

Why threads

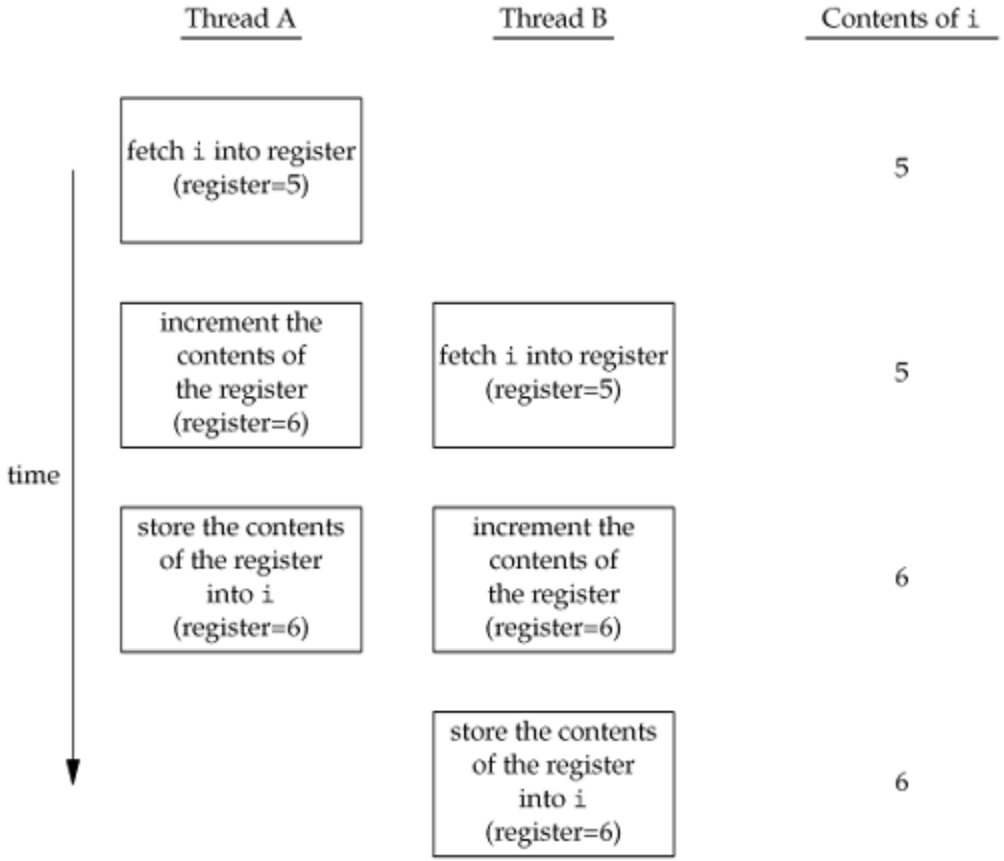
- By decomposing an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.
- The ability to communicate and share data among themselves.
- Since threads are more light-weighted than processes, they are easier (i.e., faster) to create and destroy than processes.
- Threads are useful on systems with multiple CPUs, where real parallelism is possible.

An Example of Synchronization

- Both Thread A and Thread B are trying to increment the value of variable i

- The increment operation consists of three steps:
  - Fetch the value
  - Increment
  - Store the value

- The expected ultimate value of i in memory should be 7 due to two consecutive incremental operations



```c
static int g_val=0;
void* pthread_mem(void* arg)
{
    int i=0;
    int val=0;
    while(i<500000)
    {
        val = g_val;
        i++;
        g_val=val+1;
    }
    return NULL;
}

int main()
{
    pthread_t tid1;
    pthread_t tid2;
    pthread_create(&tid1, NULL, pthread_mem, NULL);
    pthread_create(&tid2, NULL, pthread_mem, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("g_val end is :%d\n",g_val);
    return 0;
}
```

Synchronization

We want to prevent data inconsistency when working with concurrent threads of execution.

We want to be deterministic with what the outcome could be.

```
claude@ROG:~/cs214/rec07$ vim thread_test.c
claude@ROG:~/cs214/rec07$ gcc thread_test.c -lpthread
claude@ROG:~/cs214/rec07$ ./a.out
g_val end is :528417
claude@ROG:~/cs214/rec07$ ./a.out
g_val end is :487936
claude@ROG:~/cs214/rec07$ ./a.out
g_val end is :499997
claude@ROG:~/cs214/rec07$ ./a.out
g_val end is :500000
```

Synchronization

- **critical section:** A section of code that reads to or writes from shared data
- **race condition:** The potential for interleaved execution of a critical section by multiple threads
- **mutual exclusion:** A synchronization mechanism to avoid race conditions by ensuring exclusive execution of critical sections

Synchronization - locks

When one threads enters a critical section, other threads should be “locked” out from entering. They should wait until the first thread finishes and “unlocks” the critical section.

Mutexes serve as this “lock”.

- A mutex is a lock that we set before using a shared resource and unlock after we are done
- Other threads of execution cannot access the shared resource if the protecting mutex is locked
- Used to ensure synchronized access to critical section

The pthread library has an implementation of mutexes

mutex

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

- initialize a lock with attributes

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

- Tries to lock the mutex. If it is already locked, calling thread is blocked.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Same as lock(), but does not block and immediately returns instead

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

- Tries to release the mutex. A thread connot release a mutex that is did not lock

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Destroys a mutex object. Don’t destroy a mutex while its locked.

mutex

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int g_val=0;
void* pthread_mem(void* arg)
{
    int i=0;
    int val=0;
    while(i<500000)
    {
        pthread_mutex_lock(&mutex);
        val = g_val;
        i++;
        g_val=val+1;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```
claude@ROG:~/cs214/rec07$ gcc mutex_test.c -lpthread
claude@ROG:~/cs214/rec07$ ./a.out
g_val end is :1000000
claude@ROG:~/cs214/rec07$ ./a.out
g_val end is :1000000
claude@ROG:~/cs214/rec07$ ./a.out
g_val end is :1000000
claude@ROG:~/cs214/rec07$ ./a.out
g_val end is :1000000
```

Deadlock

- Deadlock is a synchronization problem where a members of a group are waiting for another to take action.

- A deadlock situation can arise if all of the following conditions hold simultaneously in a system:
  - Mutual exclusion: at least one resource must be held in a non-shareable mode. Only one process can use the resource at any given instant of time.
  - Hold and wait or resource holding: a process is currently holding at least one resource and requesting additional resources which are being held by other processes.
  - No preemption: a resource can be released only voluntarily by the process holding it.
  - Circular wait: a process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, P = {P1, P2, …, PN}, such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3 and so on until PN is waiting for a resource held by P1.
- Deadlock avoidance
  - All threads that need multiple locks should acquire them in the same order and release them in the opposite order

Condition variables

- Basically a combination of locks and signal.
- Condition variables are data structures that are used to wait for a condition to be true.
- It is an explicit queue that threads can put themselves on to wait for a condition/resource.

- Three functions:
  - `phthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mut)`
    - release lock and put thread to sleep until condition is signaled
    - it re-acquires the lock before it resume the thread
  - `pthread_cond_signal(pthread_cond_t *cond)`
    - wake up one thread that is waiting on the condition. Caller must hold the lock
  - `pthread_cond_broadcast(pthread_cond_t *cond)`
    - same as signal, but wakes all waiting threads

- Can be used for producer/consumer models

- Further reading material: https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf

Thanks