PTHREAD PROGRAMMING

UNIX Programming 2014 Fall by Euiseong Seo

Traditional View

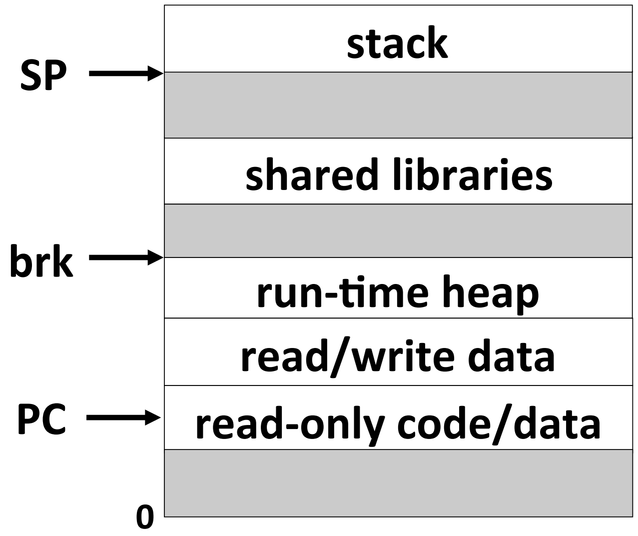
- Process = process context + address space

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

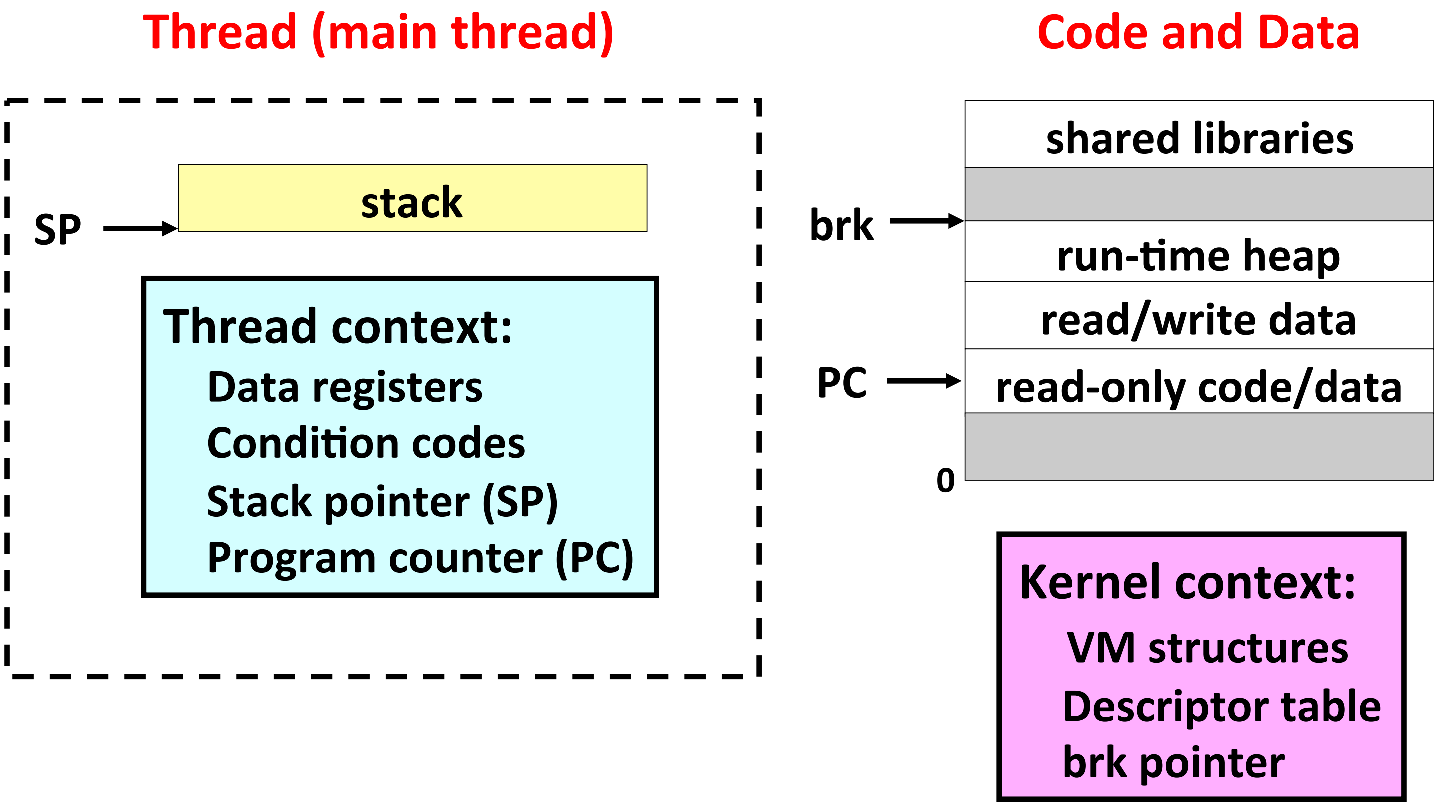
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



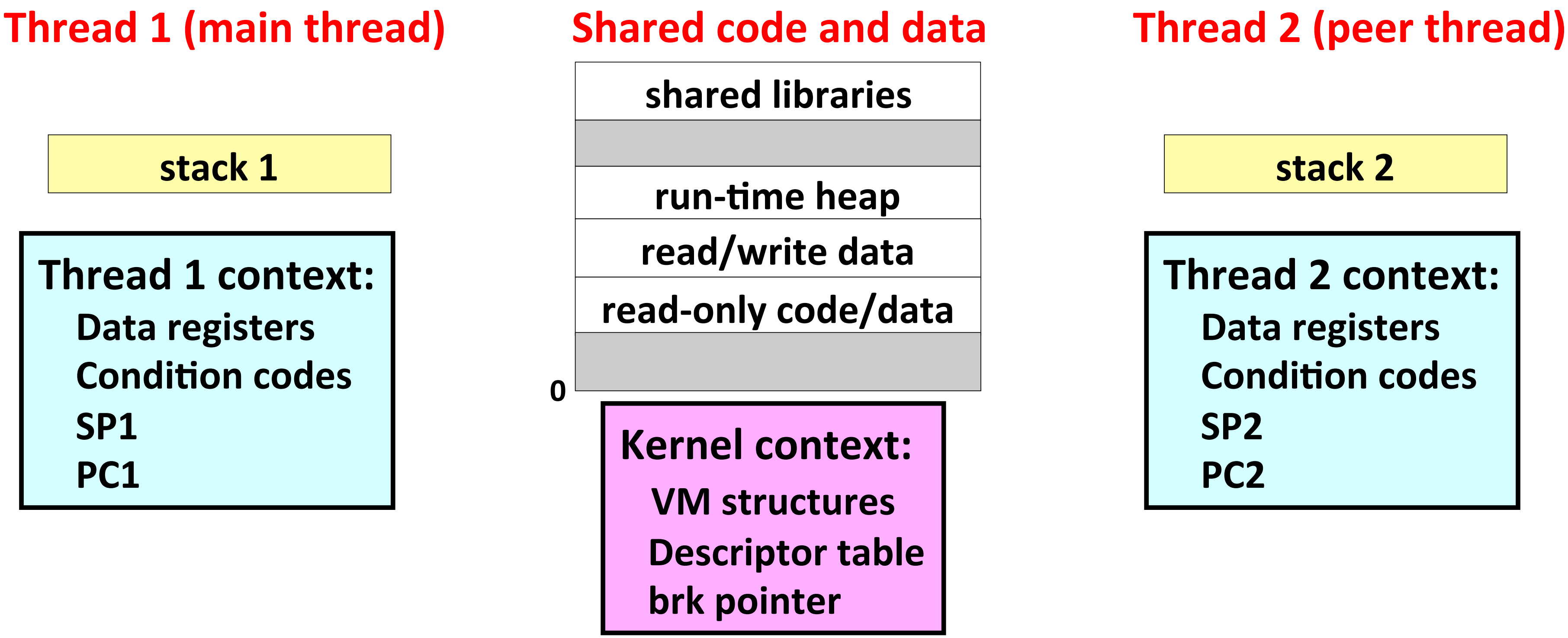
Alternate View

- Process = thread context + kernel context + address space



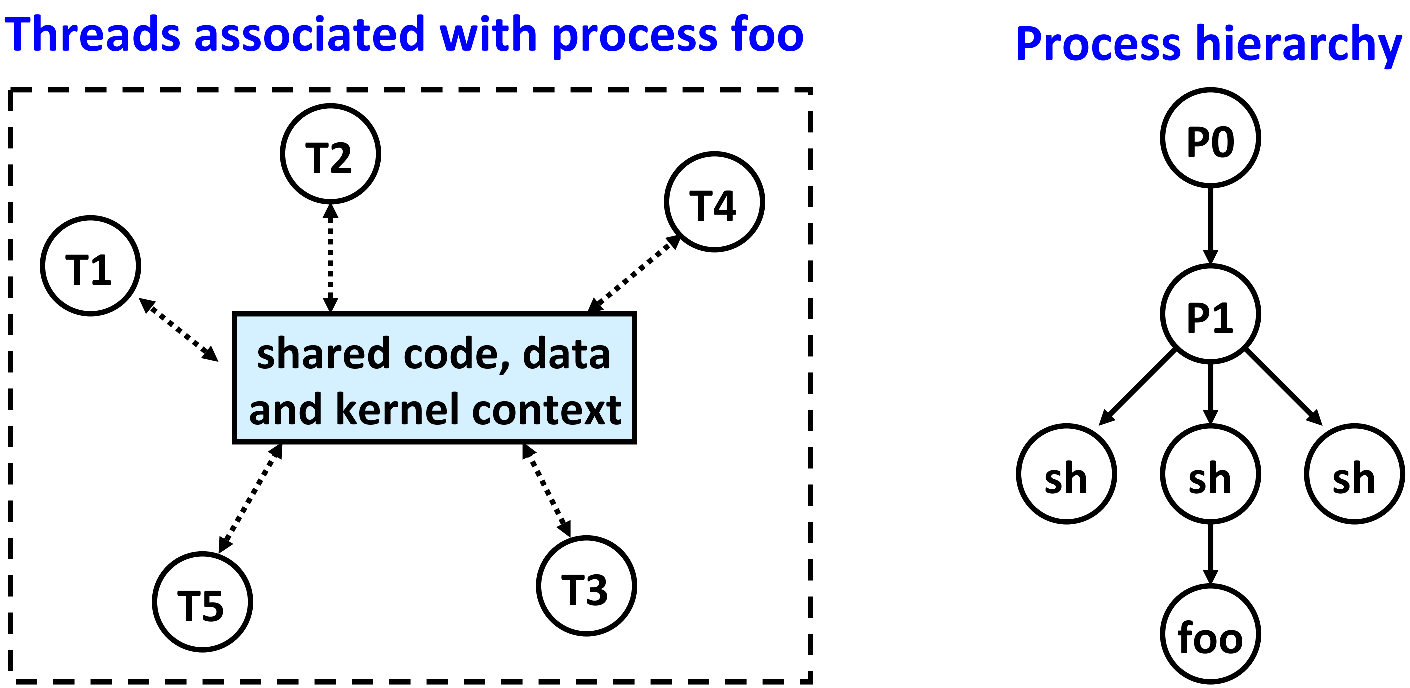
A Process with Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow (sequence of PC values)
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own thread id (TID)



Logical View of Threads

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



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
    - Linux 2.4 Kernel, 512MB RAM, 2 CPUs

      -> 1,811 forks()/second

      -> 227,611 threads/second (125x faster)

Thread-based Designs

- Pros
  - Easy to share data structures between threads
    - e.g., logging information, file cache, etc.
  - Threads are more efficient than processes

- Cons
  - Unintentional sharing can introduce subtle and hard-to-reproduce errors

Pthreads API

- ANSI/IEEE POSIX1003.1-1995 Standard
- Thread management
  - Work directly on threads – creating, terminating, joining, etc
  - Include functions to set/query thread attributes
- Mutexes
  - Provide for creating, destroying, locking and unlocking mutexes
- Condition variables
  - Include functions to create, destroy, wait and signal based upon specified variable values

Pthreads Interface

- POSIX Threads Interface
  - Creating and reaping threads
    - **pthread_create()**
    - **pthread_join()**
  - Determining your thread ID
    - **pthread_self()**
  - Terminating threads
    - **pthread_cancel()**
    - **pthread_exit()**
    - **exit** (terminates all threads), **return** (terminates current thread)
  - Synchronizing access to shared variables
    - **pthread_mutex_init()**
    - **pthread_mutex_[un]lock()**
    - **pthread_cond_init()**
    - **pthread_cond_[timed]wait()**
    - **pthread_cond_signal()**, etc.

Thread Identifiers

- Prototype

```
#include <pthread.h>

pthread_t pthread_self(void);
int pthread_equal(pthread_t t1, pthread_t t2);
```

- pthread_self() returns TID of calling thread
- Because thread IDs are opaque objects, the C language equivalence operator == should never be used to compare two thread IDs against each other

Creating Threads

- Prototype

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

- start_routine will be executed by a new thread
  - Only one argument to thread function
- TID of new thread will be stored in thread
- errno is usually not set by pthread routines

Terminating Threads

- Prototype

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

- Terminates calling thread and returns a value, retval
- retval is available to another thread in the same process that calls pthread_join()
- It does not close files
  - Any files opened inside the thread will remain open

Joining Threads

- Prototype

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

- Waits for specified thread to terminate
  - If thread has already terminated, this function returns immediately
- thread must be joinable
- retval is copy of exit status of thread

Threaded “Hello, World”

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include “pthread.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  pthread_create(&tid, NULL, thread, NULL);
  pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

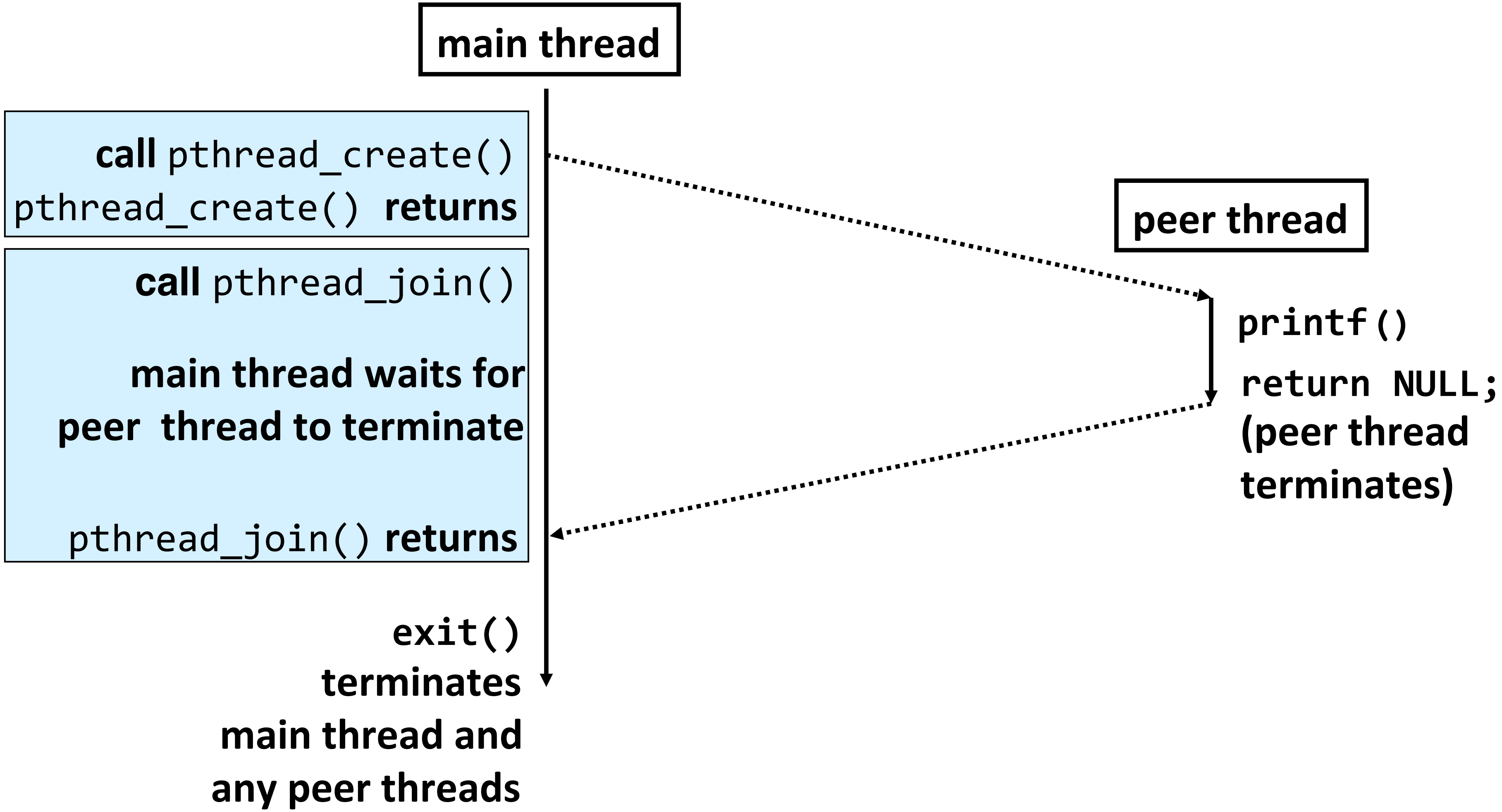
Thread attributes (usually NULL)

Thread arguments (void *p)

return value (void **p)

Threaded "Hello, World"

- Execution of threaded "hello, world"



Echo Server: Thread-based

```
int main (int argc, char *argv[])
{
   int *connfdp;
   pthread_t tid;
   . . .

   while (1) {
      connfdp = (int *)
                malloc(sizeof(int));
      *connfdp = accept (listenfd,
          (struct sockaddr *)&caddr,
          &caddrlen));

      pthread_create(&tid, NULL,
          thread_main, connfdp);
   }
}
```

```
void *thread_main(void *arg)
{
  int n;
  char buf[MAXLINE];

  int connfd = *((int *)arg);
  pthread_detach(pthread_self());
  free(arg);

  while((n = read(connfd, buf,
             MAXLINE)) > 0)
    write(connfd, buf, n);

  close(connfd);
  return NULL;
}
```

Implementation Issues (1)

- Must run "detached" to avoid memory leak
  - At any point in time, a thread is either joinable or detached
  - Joinable thread can be reaped and killed by other threads
    - Must be reaped (with pthread_join()) to free memory resources
  - Detached thread cannot be reaped or killed by other threads
    - Resources are automatically reaped on termination
    - Exit state and return value are not saved
  - Default state is joinable
    - Use pthread_detach(pthread_self()) to make detached

Implementation Issues (2)

- Must be careful to avoid unintended sharing
  - For example, what happens if we pass address of connfd to thread routine?

```
int connfd;
. . .
pthread_create(&tid, NULL, thread_main, &connfd);
. . .
```

- All functions called by a thread must be thread-safe
  - A function is said to be thread-safe or reentrant, when the function may be called by more than one thread at a time without requiring any other action on the caller's part

Cancelling Threads

- Prototype

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

- Sends a cancellation request to thread
- By default, thread behaves as it had called pthread_exit() with an argument of PTHREAD_CANCELED
  - You can ignore or defer cancellation request by pthread_setcalcelstate(3) and pthread_setcanceltype(3)
- Similarly to atexit(), a thread can arrange for functions to be called when it exists by pthread_cleanup_push() and pthread_cleanup_pop()

Detaching Threads

- Prototype

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

- Marks thread as detached
- When a detached thread terminates, its resources are automatically released back to system without need for another thread to join
- Once a thread has been attached, it can’t be joined with pthread_join() or be made joinable again

Example

```
#include <pthread.h>
#include <stdio.h>
#define NTHREADS  10

void *PrintHello(void *threadid) {
   printf("\n%d: Hello World!\n", (int)threadid);
   pthread_exit(NULL);
}

int main () {
   pthread_t tid[NTHREADS];
   int t;
   for (t = 0; t < NTHREADS; t++)
      pthread_create(&tid[t], NULL, PrintHello, (void *)t);
   for (t = 0; t < NTHREADS; t++)
      pthread_join(tid[t], NULL);
}
```

Synchronization

- When one thread can modify a variable that other threads can read or modify, we need to synchronize threads to ensure that they don't use an invalid value

Thread A	Thread B	Contents of i
		5
fetch i into register		5
increase i in register	fetch i into register	5
store i into memory	increase i in register	6
	store i into memory	6

Mutex

- A mutex is basically a lock that we set before accessing a shared resource and release when we're done
- If a mutex is set, any other thread that tries to set it will block until it is released
- If more than one thread is blocked when we unlock the mutex, then all threads blocked on the lock will be made runnable, and the first one to run will be able to set the lock
- A mutex is represented by pthread_mutex_t data type

Mutex

- Mutex initialization

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

- Setting and Unsetting a Mutex

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

Mutex Example

```
int deposit(int amount)
{
  int balance;

  balance = get_balance();
  balance += amount;
  put_balance(balance);
  return balance;
}
```

```
int withdraw(int amount)
{
  int balance;

  balance = get_balance();
  balance -= amount;
  put_balance(balance);
  return balance;
}
```

T1 executes deposit(100)

```
balance = get_balance();
balance += 100;



put_balance(balance);
```

T2 executes withdraw(300)

```
balance = get_balance();
balance -= 300;
put_balance(balance);
```

Mutex Example

```c
pthread_mutex_t m =
     PTHREAD_MUTEX_INITIALIZER;

int deposit(int amount)
{
  int balance;

  pthread_mutex_lock(&m);

  balance = get_balance();
  balance += amount;
  put_balance(balance);

  pthread_mutex_unlock(&m);

  return balance;
}
```

```c
int withdraw(int amount)
{
  int balance;

  pthread_mutex_lock(&m);

  balance = get_balance();
  balance -= amount;
  put_balance(balance);

  pthread_mutex_unlock(&m);

  return balance;
}
```

Other Synchronization

- Reader-writer locks
  - Multiple readers are allowed
  - Provides higher degree of parallelism
- Condition variables
  - Used with mutexes, condition variables allow threads to wait in a race-free way for arbitrary conditions to occur
- Spin locks
  - Basically the same as mutexes
  - Threads will not be descheduled by blocking
- Barriers
  - To coordinate multiple threads working parallel
  - A barrier allows each thread to wait until all cooperating threads have reached the same point