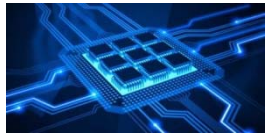
CSE320 System Fundamentals II
Threads

YoungMin Kwon

Multi-threaded Programs

- Why multi-threaded programming
  - To capture the logical structure of a program
  - Exploit parallel hardware for speed



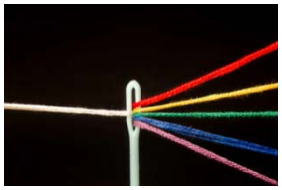
- Demo: without multi-threaded programming

`https://www.youtube.com/watch?v=MNhubpzhs-Q`

Threads

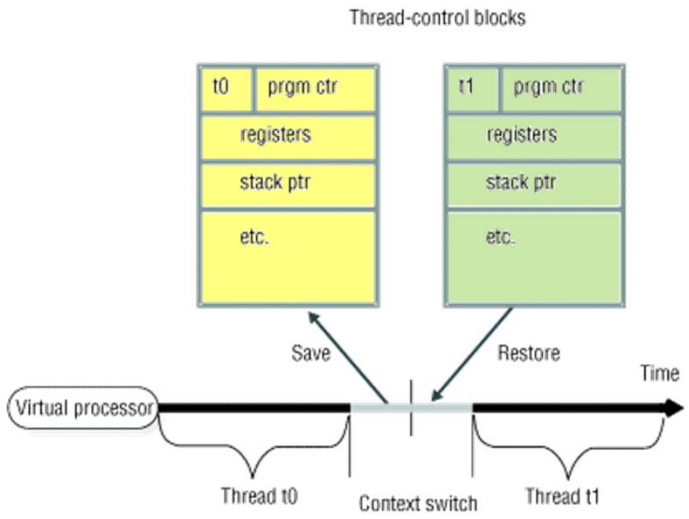
- A thread is a logical flow that runs in the context of a process

- Each process begins life as a single thread called the main thread
- Later, the main thread creates peer threads



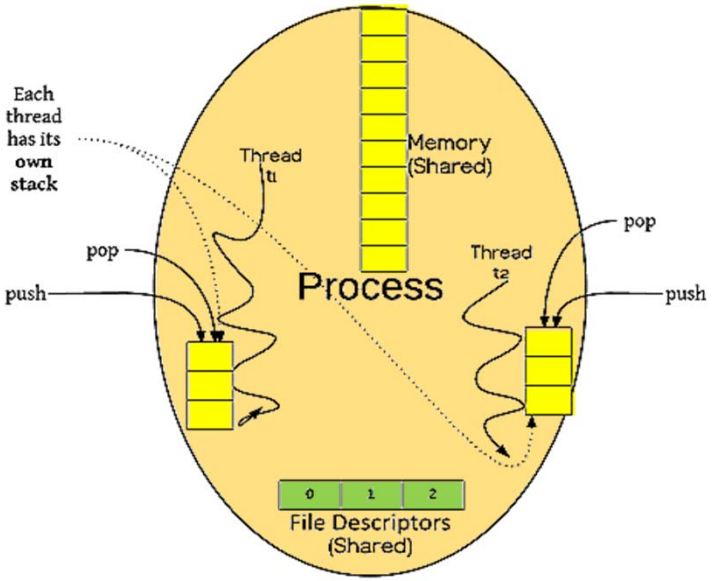
Threads

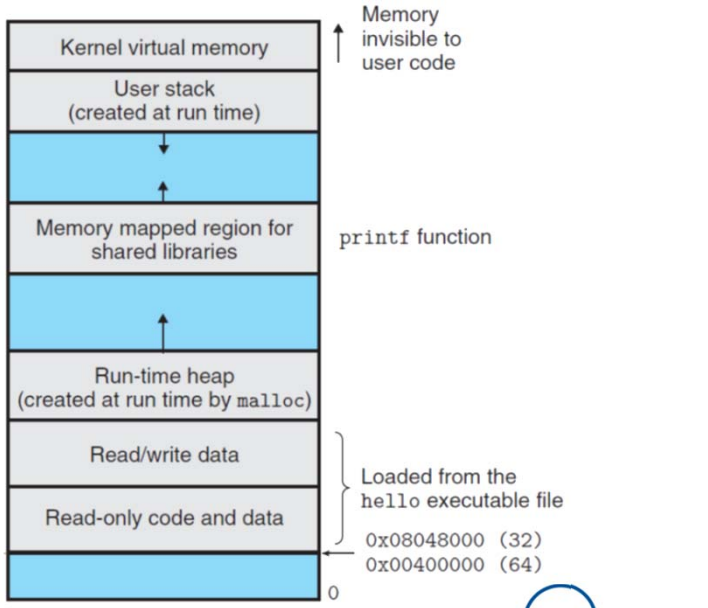
- Each thread has its own thread context
  - Thread ID (TID),
  - Stack, Stack Pointer
  - Program Counter,
  - General-purpose registers, Condition codes (Flags)



Threads

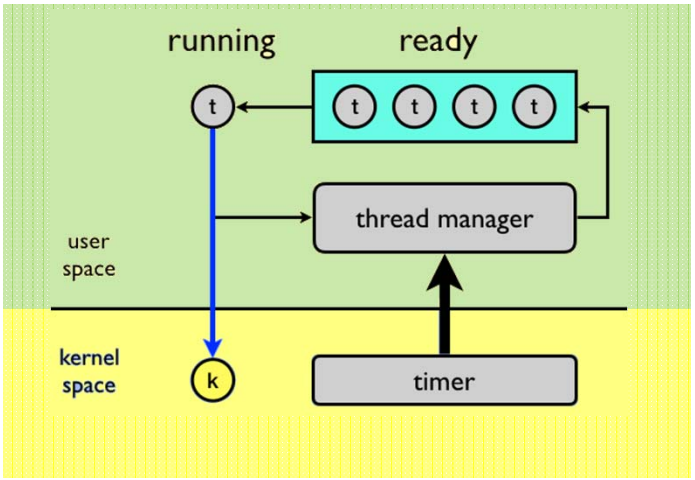
- Multiple threads run in the context of a single process
  - They share the entire virtual address space of the process
  - Code, Heap, Shared libraries, Open files

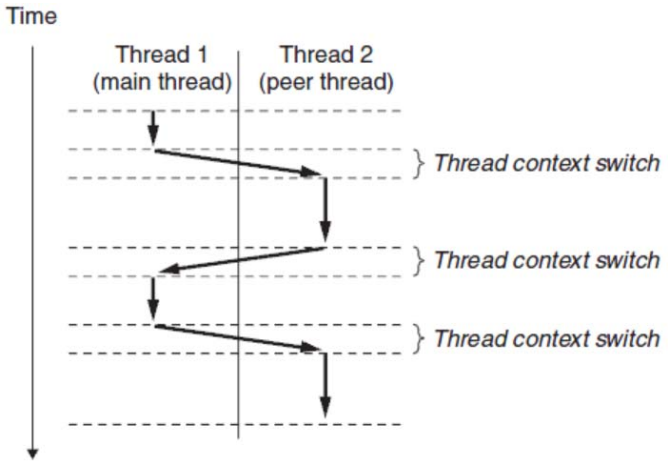




Threads

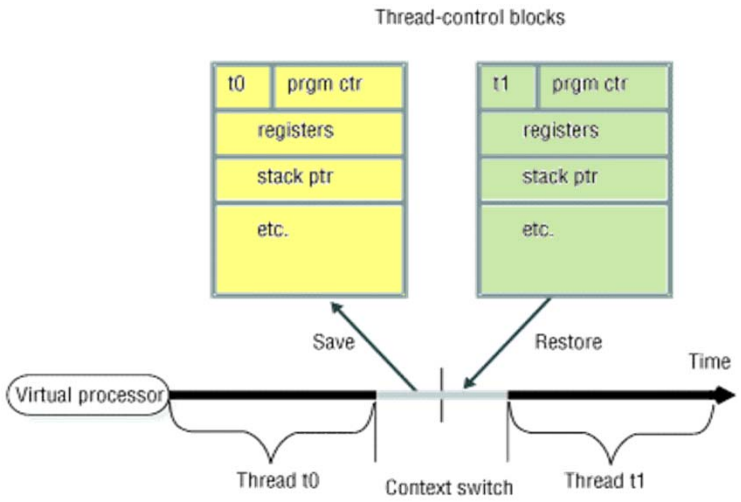
- Thread scheduling is done by the kernel
  - Slow system calls like read or sleep
  - Interrupted by the system timer





Threads

- A thread context is much smaller than a process context ⇒ context switch is faster



Threads

- Threads associated with a process form a pool of peers
  - No rigid parent-child hierarchy
  - A thread can kill any of its peers
  - A thread can wait for any of its peers to terminate
  - Each peer can read/write the same shared data

POSIX Threads (Pthreads)

- A standard interface for manipulating threads from C programs
- Defines about 60 functions that allow programs
  - to create, kill, and reap threads
  - to share data safely with peer threads
  - to notify peers about changes in the system state

Creating Threads

```
#include <pthread.h>
typedef void *(func)(void *);

// To create a new thread
int pthread_create(pthread_t *tid,
                   pthread_attr_t *attr,
                   func *f,
                   void *arg);

// To get the thread id of its own
pthread_t pthread_self(void);
```

Terminating Threads

```
#include <pthread.h>

void pthread_exit(void *thread_return);
// Terminate explicitly
// If main thread calls pthread_exit,
// it will wait for all other peer threads to
// terminate, terminate itself, and terminate
// the process

int pthread_cancel(pthread_t tid);
// Terminate the thread with the ID tid
```

Reaping Terminated Threads

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

// - blocks until thread pid terminates,
// - update thread_return to point to the return
//   value of the thread routine,
// - reap the memory resource

// - unlike waitpid, there is no way to wait for
//   an arbitrary thread to terminate
```

```c
#include <pthread.h>
#include <stdio.h>

void *thread_func(void *vargp) {
    printf("%s\n", (char*)vargp);
    pthread_exit("world");
    //return "world";
    return "";
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread_func, "hello");

    void *ret;
    pthread_join(tid, &ret);
    printf("%s\n", (char*)ret);
    return 0;
}
```

Detaching Threads

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

// - For a joinable thread (default) its
//   memory resource is not freed until the
//   thread is reaped

// - A detached thread cannot be reaped or
//   killed by other threads
// - Its memory resources are automatically
//   freed when it terminates
```

Initializing Threads

```
#include <pthread.h>
pthread_once once_control = PHTREAD_ONCE_INIT;
int pthread_once(pthread_once *once_control,
                 void (*init_routine)(void));

// - Useful to initialize shared global
//   variables dynamically
// - init_routine is called only once even if
//   pthread_once is called multiple times
```

```c
//charcount.c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void* charcount(void *vargp);

int main() {
    pthread_t tid = pthread_self();
    printf("main: %u\n", (unsigned int)tid);

    while(1) {
        char str[100];
        pthread_t tid;

        scanf("%99s", str);
        if(strcmp(str, "quit") == 0)
            break;
        pthread_create(&tid, NULL, charcount, strdup(str));
    }
    return 0;
}
```

```c
void* charcount(void *vargp) {
    char *str = (char*)vargp;
    int count[256] = {0,};
    pthread_t tid = pthread_self();
    int i;

    pthread_detach(tid);
    printf("server %u\n", (unsigned int)tid);

    for(i = 0; str[i]; i++)
        count[ str[i] ]++; // Counting the occurrence of each char

    for(i = 0; i < 256; i++) {
        if(count[i] > 0)
            printf("%u: '%c': %d\n", (unsigned int)tid, i, count[i]);
    }

    free(str);
    return NULL;
}
```

```
$ gcc charcount.c -pthread
```

Shared Variables

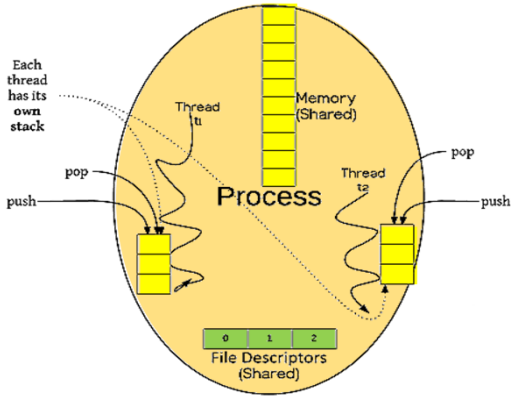
- One of the merits of using threads is sharing variables between threads
  - A variable is shared iff multiple threads reference an instance of the variable
  - However, ***sharing variables (resources) is tricky***

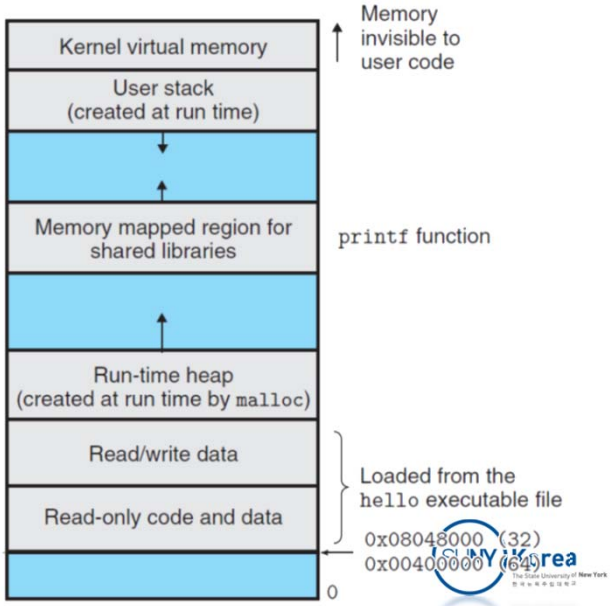
```c
#include <pthread.h>
#include <stdio.h>
char **ptr;                         // .bss
void *thread(void *vargp) {
    long myid = (long)vargp;        // stack
    static int cnt = 1;             // .data
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}

int main() {
    long i;
    pthread_t tid;
    char *msg[2] = {"hello", "world"};

    ptr = msg;  // copy a local msg var to a global ptr var
    for(i = 0; i < 2; i++)
        pthread_create(&tid, NULL, thread, (void*)i);
    pthread_exit(NULL);
}
```

```
$ a.out
[0]: hello (cnt=2)
[1]: world (cnt=3)
```

Thread Memory Model

- Registers are never shared
- Virtual memory is always shared
- Each thread has its own stack, but stack space is not protected





```c
#include <pthread.h>
#include <stdio.h>

void *thread(void *vargp) {
    int a;   //allocated in each thread's stack
    printf("%ld: &a = %p\n", pthread_self(), &a);
    pthread_exit(NULL);
    return NULL;
}

int main() {
    pthread_t tid;
    int i = 0;
    for(i = 0; i < 10; i++)
        pthread_create(&tid,
                       NULL,
                       thread,
                       NULL);

    pthread_exit(NULL);
    return 0;
}
```

```
$ ./a.out
140318187357952: &a = 0x7f9e5fb67edc
140318153787136: &a = 0x7f9e5db63edc
140318178965248: &a = 0x7f9e5f366edc
140318170572544: &a = 0x7f9e5eb65edc
140318145394432: &a = 0x7f9e5d362edc
140318137001728: &a = 0x7f9e5cb61edc
140318162179840: &a = 0x7f9e5e364edc
140318128609024: &a = 0x7f9e5c360edc
140318120216320: &a = 0x7f9e5bb5fedc
140318111823616: &a = 0x7f9e5b35eedc
```

Synchronizing Threads

- Sharing variables are convenient
- But, be careful with synchronization errors

```c
#include <pthread.h>
#include <stdio.h>

volatile long cnt = 0;

void* thread(void *vargp) {
    long i, niters = *((long*)vargp);
    for(i = 0; i < niters; i++)
        cnt++;
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    long n = 100000;

    pthread_create(&tid1, NULL, thread, &n);
    pthread_create(&tid2, NULL, thread, &n);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("cnt = %ld\n", cnt);
    return 0;
}
```

Synchronizing Threads

C code for thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

Asm code for thread i

```
    movq   (%rdi), %rcx        } Hi : Head
    testq  %rcx,%rcx
    jle    .L2
    movl   $0, %eax
.L3:
    movq   cnt(%rip),%rdx      } Li : Load cnt
    addq   $1, %rdx            } Ui : Update cnt
    movq   %rdx, cnt(%rip)     } Si : Store cnt
    addq   $1, %rax            } Ti : Tail
    cmpq   %rcx, %rax
    jne    .L3
.L2:
```

- H_i: instructions at the head of the loop
- L_i: load the variable cnt into %rdx
- U_i: update (increment) %rdx
- S_i: store updated %rdx into cnt
- T_i: instructions at the tail of the loop

Synchronizing Threads

- Correct Ordering

Step	Thread	Instr	%rdx$_1$	%rdx$_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	1	S_1	1	—	1
5	2	H_2	—	—	1
6	2	L_2	—	1	1
7	2	U_2	—	2	1
8	2	S_2	—	2	2
9	2	T_2	—	2	2
10	1	T_1	1	—	2

Synchronizing Threads

- Incorrect Ordering

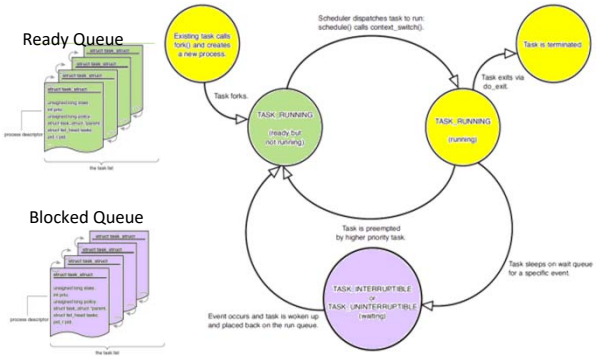
Step	Thread	Instr	%rdx$_1$	%rdx$_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	2	H_2	—	—	0
5	2	L_2	—	0	0
6	1	S_1	1	—	1
7	1	T_1	1	—	1
8	2	U_2	—	1	1
9	2	S_2	—	1	1
10	2	T_2	—	1	1

Semaphores

- Semaphores are used to synchronize different executions of threads

Semaphores

- Semaphore
  - A nonnegative global variable
  - Manipulated by P and V operations



- P(s):
  - if s > 0, then decrement s and return
  - if s = 0, suspend the thread until s is increased by V from other thread. After restarting decrement s and return
- V(s):
  - increase s by 1.
  - if any threads are blocked by P, V restarts exactly one of the blocked threads

Semaphores

```
#include <semaphore.h>

int sem_init(sem_t *s,     // Semaphore
             int pshared, // 0: between threads;
                          // otherwise, between processes
             unsigned int value); // initial value

int sem_destroy(sem_t *s); //destroy the semaphore s

int sem_wait(sem_t *s);    // P(s)

int sem_post(sem_t *s);    // V(s)
```

Semaphores

- Binary semaphore
  - To provide mutual exclusion (mutex)
  - Semaphore has values 0 and 1
  - P operation is called locking
  - V operation is called unlocking
- Counting semaphore
  - Used as a counter for a set of available resources

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

sem_t mutex;
volatile long count = 0;
void* thread(void *vargp) {
    long i, n = *((long*)vargp);

    for(i = 0; i < n; i++) {
        sem_wait(&mutex);
        count++;
        sem_post(&mutex);
    }

    return NULL;
}
```

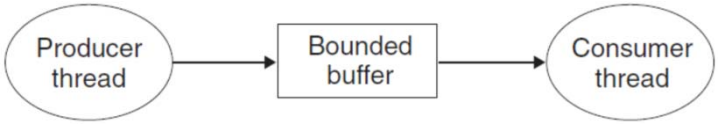
```c
int main() {
    pthread_t tid1, tid2;
    long n = 100000;

    sem_init(&mutex, 0/*pshared*/, 1/*value*/);

    pthread_create(&tid1, NULL, thread, &n);
    pthread_create(&tid2, NULL, thread, &n);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    sem_destroy(&mutex);

    printf("count = %ld\n", count);
    return 0;
}
```

Semaphores

- Counting Semaphores
  - To schedule shared resources
  - Notify other threads that some program states have changed
- Example usages
  - Producer-consumer problem
  - Readers-writers problem

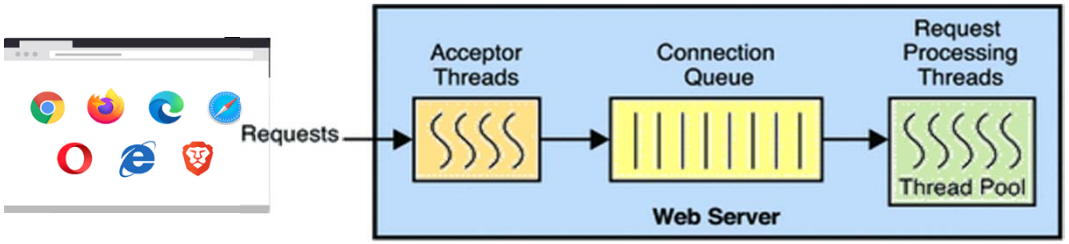
Producer-Consumer Problem

- A producer and a consumer thread share a bounded buffer with n slots



  - The producer creates items and adds them to the buffer
  - The consumer removes items from the buffer and consumes (uses) them

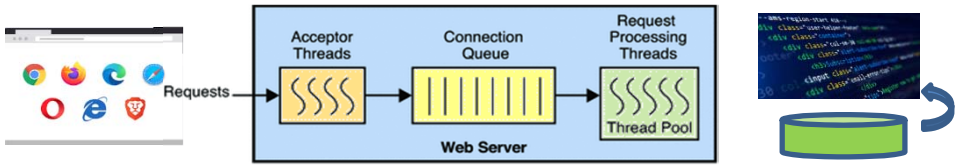
- E.g. Web server:

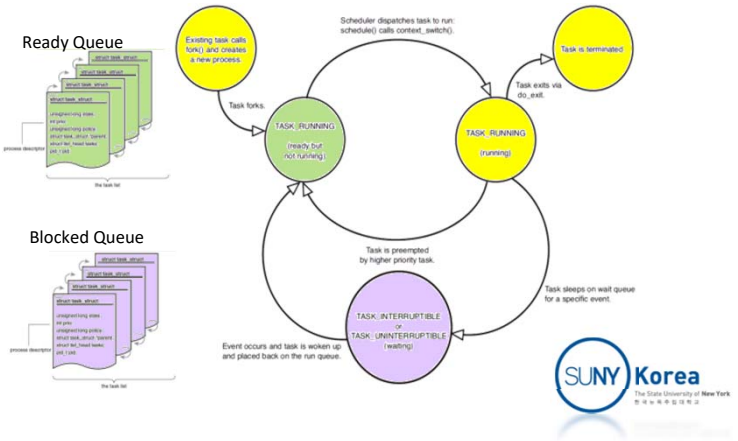


Producer-Consumer Problem

- Need a mutual exclusion to access the shared buffer

- Need to schedule the access to the buffer
  - If the buffer is full, the producer needs to wait
  - If the buffer is empty, the consumer needs to wait





```c
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
typedef struct { //shared buffer
    int *buf;
    int capacity, head, tail;
    sem_t mutex;
    sem_t slots;
    sem_t items;
} sbuf_t;

void sbuf_init(sbuf_t* sp, int n) {
    sp->buf = (int*) calloc(n, sizeof(int));
    sp->capacity = n;
    sp->head = sp->tail = 0;

    sem_init(&sp->mutex, 0, 1);
    sem_init(&sp->slots, 0, n);
    sem_init(&sp->items, 0, 0);
}
```

```c
void sbuf_deinit(sbuf_t *sp) {
    free(sp->buf);
    sem_destroy(&sp->mutex);
    sem_destroy(&sp->slots);
    sem_destroy(&sp->items);
}


int sbuf_size(sbuf_t *sp) {

    //access the buffer after acquiring the lock
    sem_wait(&sp->mutex);

    int n = (sp->head - sp->tail + sp->capacity) % sp->capacity;

    //release the lock
    sem_post(&sp->mutex);

    return n;
}
```

```c
void sbuf_insert(sbuf_t *sp, int item) {
    sem_wait(&sp->slots); // wait while the buffer is full
    sem_wait(&sp->mutex); // acquire the lock to access buffer

    sp->head = (sp->head + 1) % sp->capacity;
    sp->buf[sp->head] = item;

    sem_post(&sp->mutex); // release the lock
    sem_post(&sp->items); // wake up consumer if it is suspended
}


int sbuf_remove(sbuf_t *sp) {
    sem_wait(&sp->items); // wait while the buffer is empty
    sem_wait(&sp->mutex); // acquire the lock to access buffer

    sp->tail= (sp->tail + 1) % sp->capacity;
    int item = sp->buf[sp->tail];

    sem_post(&sp->mutex); // release the lock
    sem_post(&sp->slots); // wake up producer if it is suspended
    return item;
}
```

```c
void* producer(void* vargp) {
    sbuf_t *sp = (sbuf_t*)vargp;
    int i, j;
    for(i = 0; i < 100; i++) {
        long s = 0;
        for(j = 0; j < 10000; j++)
            s += j, sbuf_insert(sp, j);
        printf("producer: sum: %ld, size: %d\n", s, sbuf_size(sp));
    }
    pthread_exit(NULL);
}

void* consumer(void* vargp) {
    sbuf_t *sp = (sbuf_t*)vargp;
    int i, j;
    for(i = 0; i < 100; i++) {
        long s = 0;
        for(j = 0; j < 10000; j++)
            s += sbuf_remove(sp);
        printf("consumer: sum: %ld, size: %d\n", s, sbuf_size(sp));
    }
    pthread_exit(NULL);
}
```

```c
int main() {
    pthread_t tid_p, tid_c;
    sbuf_t sb;
    sbuf_init(&sb, 15000);

    pthread_create(&tid_p, NULL, producer, &sb);
    pthread_create(&tid_c, NULL, consumer, &sb);

    pthread_join(tid_p, NULL);
    pthread_join(tid_c, NULL);

    sbuf_deinit(&sb);
    return 0;
}
```

Readers-Writers Problem

- Concurrent threads accessing a shared object
  - Reader: threads that only read the data
  - Writer: threads that only modify the data

- First readers-writers problem (favors readers)
  - No readers keep waiting unless a writer has already been granted permission to update the object

  - Second readers-writers problem (favors writers)

```c
// First Readers-Writers problem
//
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

typedef struct {
    sem_t mutex;
    sem_t wlock;
    int readcount;
} rwlock;

typedef struct {
    int data;
    int copy;
    rwlock lock;
} object;
```

```c
void rwlock_init(rwlock *lock) {
    sem_init(&lock->mutex, 0, 1);
    sem_init(&lock->wlock, 0, 1);
    lock->readcount = 0;
}

void rwlock_deinit(rwlock *lock) {
    sem_destroy(&lock->mutex);
    sem_destroy(&lock->wlock);
}
```

```
void acquire_reader_lock(rwlock *lock) {
    //acquire mutex to access lock
    sem_wait(&lock->mutex);

    lock->readcount++;

    //if this is the first reader, wait until any possible writer
    //finishes writing and block future writers
    if(lock->readcount == 1)
        sem_wait(&lock->wlock);

    //release the access lock
    sem_post(&lock->mutex);
}
```

```
void release_reader_lock(rwlock *lock) {
    //acquire mutex to access lock
    sem_wait(&lock->mutex);

    lock->readcount--;

    //if this is the last reader, unblock any writers waiting
    if(lock->readcount == 0)
        sem_post(&lock->wlock);

    //release the access lock
    sem_post(&lock->mutex);
}
```

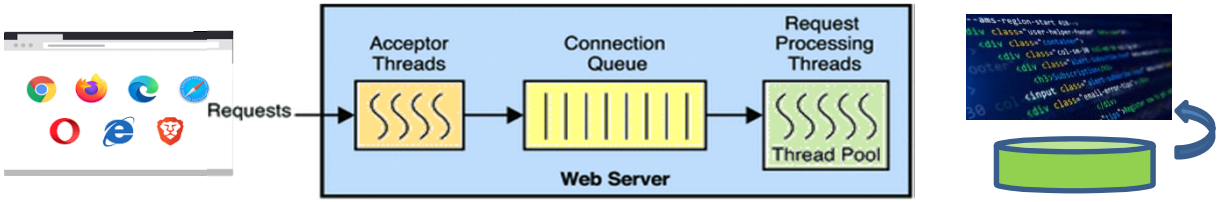
```
void acquire_writer_lock(rwlock *lock) {
    //acquire the write lock
    sem_wait(&lock->wlock);
}


void release_writer_lock(rwlock *lock) {
    //release the write lock
    sem_post(&lock->wlock);
}
```

```
void* reader(void *vargp) {
    object* pobj = (object*)vargp;
    int i;
    for(i = 0; i < 10000; i++) {
        acquire_reader_lock(&pobj->lock);
        int data = pobj->data;
        int copy = pobj->copy;
        release_reader_lock(&pobj->lock);
        printf("R_%d: data: %d, copy: %d\n", i, data, copy);
    }
}

void* writer(void *vargp) {
    object* pobj = (object*)vargp;
    int i;
    for(i = 0; i < 10000; i++) {
        acquire_writer_lock(&pobj->lock);
        int data = pobj->data = i % 10;
        int copy = pobj->copy = pobj->data;
        release_writer_lock(&pobj->lock);
        printf("W_%d: data: %d, copy: %d\n", i, data, copy);
    }
}
```

```
int main() {
    pthread_t tid[3];
    object obj;
    obj.data = obj.copy = 0;
    rwlock_init(&obj.lock);

    pthread_create(tid+0, 0, reader, &obj);
    pthread_create(tid+1, 0, reader, &obj);
    pthread_create(tid+2, 0, writer, &obj);

    pthread_join(tid[0]);
    pthread_join(tid[1]);
    pthread_join(tid[2]);

    rwlock_deinit(&obj.lock);
}
```

Programming Assignment 10

- Implement a simple http server



  - Download http_server.zip
  - Implement all TODO parts

- Due date: TBD

Programming Assignment 10

- To build
  - Run make
- To test
  - Copy the files in pages to a directory
  - Run hs with the directory name
  - Load browsers.html from a browser with port 8080
    - E.g.: http://127.0.0.1:8080/browsers.html
  - Reload the page

```c
#ifndef __SERVER__
#define __SERVER__

#include "task.h"
#include "worker.h"

#define SERV_PORT   8080
#define NR_WORKER   4

typedef struct server {
    task_queue_t *task_q;
    worker_t *worker[NR_WORKER];
    char *root_dir;

    void (*start_workers)(struct server *self);
    void (*stop_workers )(struct server *self);
    void (*run          )(struct server *self);
    void (*destroy      )(struct server *self);
} server_t;

extern server_t *make_server(char *root_dir);

#endif
```

```
#ifndef __WORKER__
#define __WORKER__
…
typedef struct task_queue {
    list_t head;    //list of tasks
    sem_t lock;     //mutex
    sem_t nr_task;  //number of tasks

    void    (*destroy)(struct task_queue *self);
    void    (*add    )(struct task_queue *self, task_t *task);
    task_t *(*remove )(struct task_queue *self);
} task_queue_t;

typedef struct worker {
    int id;                 //worker id
    pthread_t tid;          //thread id of the worker
    int done;               //if true exit the thread
    task_queue_t *task_q;   //task queue

    void  (*destroy)(struct worker *self);
    void  (*start  )(struct worker *self);
    void  (*stop   )(struct worker *self);
    void *(*work   )(void *_self); //thread func
} worker_t;

extern task_queue_t *make_task_queue();
extern worker_t *make_worker(task_queue_t *task_q);

#endif
```

```c
#ifndef __TASK__
#define __TASK__

#include "list.h"

typedef struct task {
    int id;             //task id
    int fd;             //client socket
    char *root_dir;     //root dir of pages
    char *method;       //we handle the GET method only
    char *url;          //the url of the requested resource
    list_t list;        //link of the next/prev task

    void (*read_request   )(struct task *self);
    void (*write_response )(struct task *self);
    void (*sample_response)(struct task *self);
    void (*destroy        )(struct task *self);
} task_t;

extern task_t *make_task(int fd, char *root_dir);

#endif
```

Expected result

```
$ ./hs pages
[task   0] from client: 10.12.21.87:57336
[worker 0] processing task 0
GET /browsers.html HTTP/1.1
[task   1] from client: 10.12.21.87:57592
[worker 1] processing task 1
GET /chrome.png HTTP/1.1
[task   2] from client: 10.12.21.87:58360
[worker 2] processing task 2
GET /edge.png HTTP/1.1
[task   3] from client: 10.12.21.87:59128
[task   4] from client: 10.12.21.87:58872
[worker 3] processing task 3
[worker 0] processing task 4
[task   5] from client: 10.12.21.87:58616
[worker 1] processing task 5
GET /safari.png HTTP/1.1
GET /firefox.png HTTP/1.1
GET /vivaldi.png HTTP/1.1
…
[task   7] from client: 10.12.21.87:61944
[task   8] from client: 10.12.21.87:62200
[worker 1] processing task 7
[worker 0] processing task 8
GET /browsers.html HTTP/1.1
GET /chrome.png HTTP/1.1
[task   9] from client: 10.12.21.87:62456
[worker 0] terminated
[worker 1] terminated
[worker 2] terminated
[worker 3] terminated
```

Server is terminated after reloading