



Multithread Models

- **Many-to-One:**
 - many user threads to one kernel thread.
 - Example: Solaris
- **One-to-One:**
 - each user thread is mapped to a kernel thread
 - Example: Linux & Windows family (at least to XP, not sure about Win7)
- **Many-to-Many:**
 - Many user threads to smaller or equal # of kernel thread.
 - Example: IRIX, HP-UX, and Tru64 UNIX (Solaris Prior to v9)

Why Threads?

- **Responsiveness** - web browsing in one thread, loading images in another
- **Resource sharing** - allow several thread of activity in same address space
- **Economy** - memory & resources for process creation is costly while threads share resources of a process (Solaris is 30x slower creating process than a thread)
- **Scalability** - In multiprocessor, threads can run parallel on diff processors, single-thread process can only run on one processor regardless of how many processors exist

Linux Threads

- Linux does not distinguish between processes and threads - It uses the more generic term "tasks".
- The traditional `fork()` system call completely duplicates a process (task), as described earlier.
- An alternative system call, `clone()` allows for varying degrees of sharing between the parent and child tasks, controlled by flags such as those shown in the following table:

flag	Meaning
<code>CLONE_FS</code>	File-system information is shared
<code>CLONE_VM</code>	The same memory space is shared
<code>CLONE_SIGHAND</code>	Signal handlers are shared
<code>CLONE_FILES</code>	The set of open files is shared

Linux Threads

- Calling `clone()` with no flags set is equivalent to `fork()`. Calling `clone()` with `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND`, and `CLONE_FILES` is equivalent to creating a thread, as all of these data structures will be shared.
- Linux implements this using a structure **`task_struct`**, which essentially provides a level of indirection to task resources. When the flags are not set, then the resources pointed to by the structure are copied, but if the flags are set, then only the pointers to the resources are copied, and hence the resources are shared. (Think of a deep copy versus a shallow copy in OO programming.)

Multi Thread Context Switch (TCS)

Linux has a unique implementation of threads. To the Linux kernel, there is *no* concept of a thread.

Linux implements all threads as standard processes. The Linux kernel does not provide any special scheduling semantics or data structures to represent threads.

Instead, a thread is merely a process that shares certain resources with other processes.

Each thread has a unique `task_struct` and appears to the kernel as a normal process (which just happens to share resources, such as an address space, with other processes)

Multithreading Models

Process vs Threads

Process

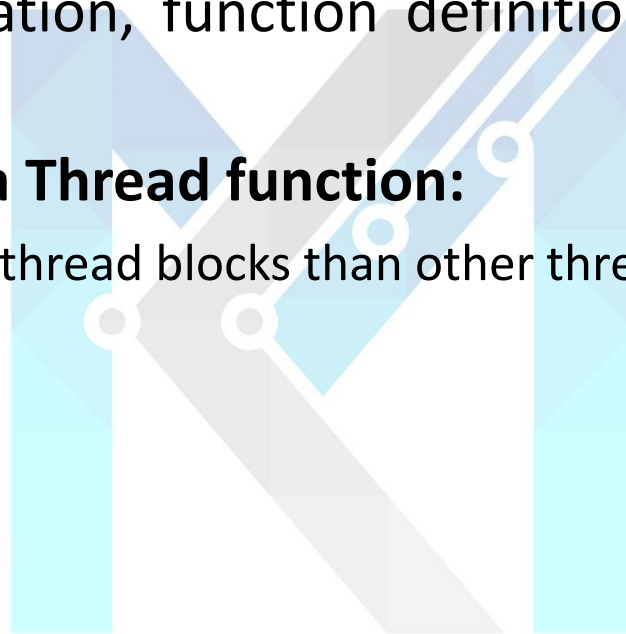
- Process is an execution of a program. Program by itself is not a process.
- Process is a parallel execution.
- Whenever you are creating a process text segment, data segment, stack segments are created.

Thread

- Thread is a smallest unit of execution (Light weight process).
- Thread is a sequence of control with in a process.
- Whenever you are creating a thread it will create only stack segment and share text and data segment. i.e., why we are saying thread is smallest unit of execution.

Threads vs Functions

- Implementation thread is almost same as a function like function declaration, function definition except that function invocation.
- **How to invoke a Thread function:**
 - Whenever one thread blocks than other thread runs.



Drawbacks of Thread

- Debugging a multithreaded program is much harder than debugging a single-threaded one, because the interactions between the threads are very hard to control.
- Writing multithreaded programs requires very careful design.
- A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine, as there are only so many CPU cycles to be had, though if nothing else.

Pthreads



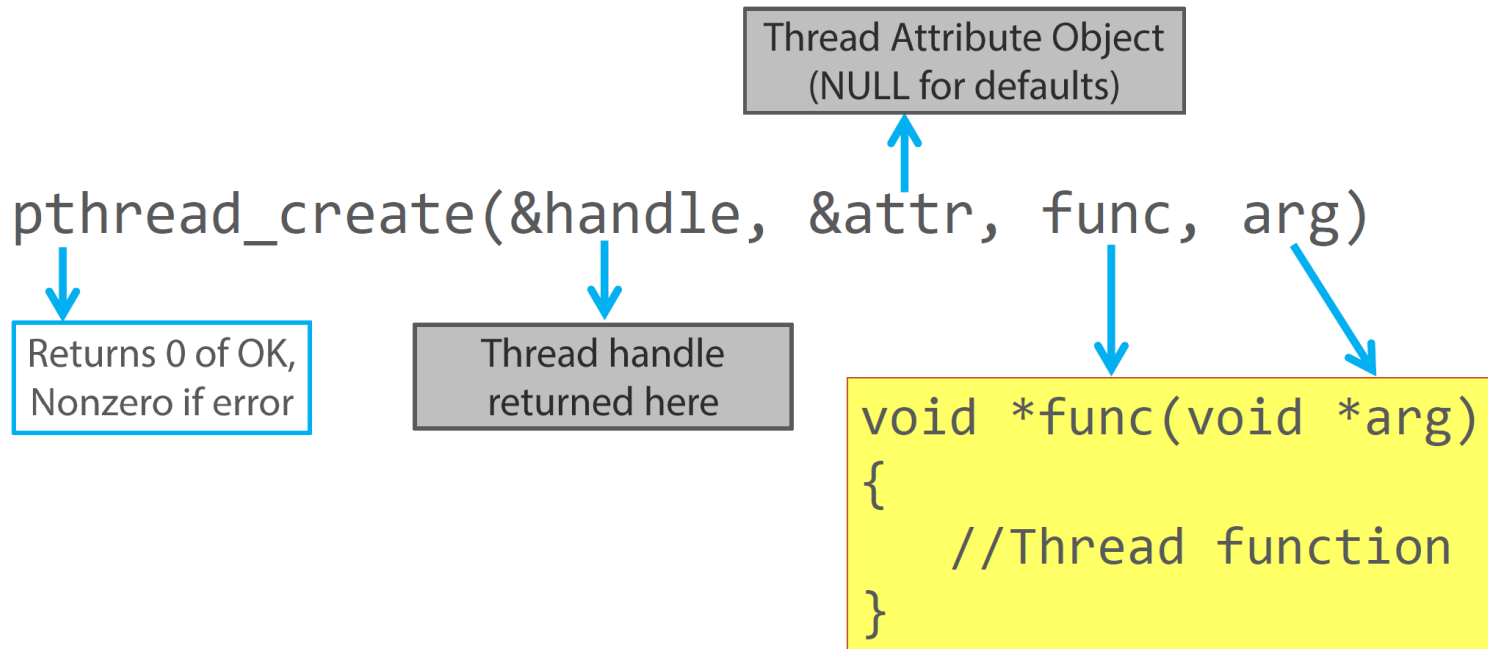
POSIX 1003.1c



A standardised set of C library routines
for thread creation and management

— Upwards of 60 functions

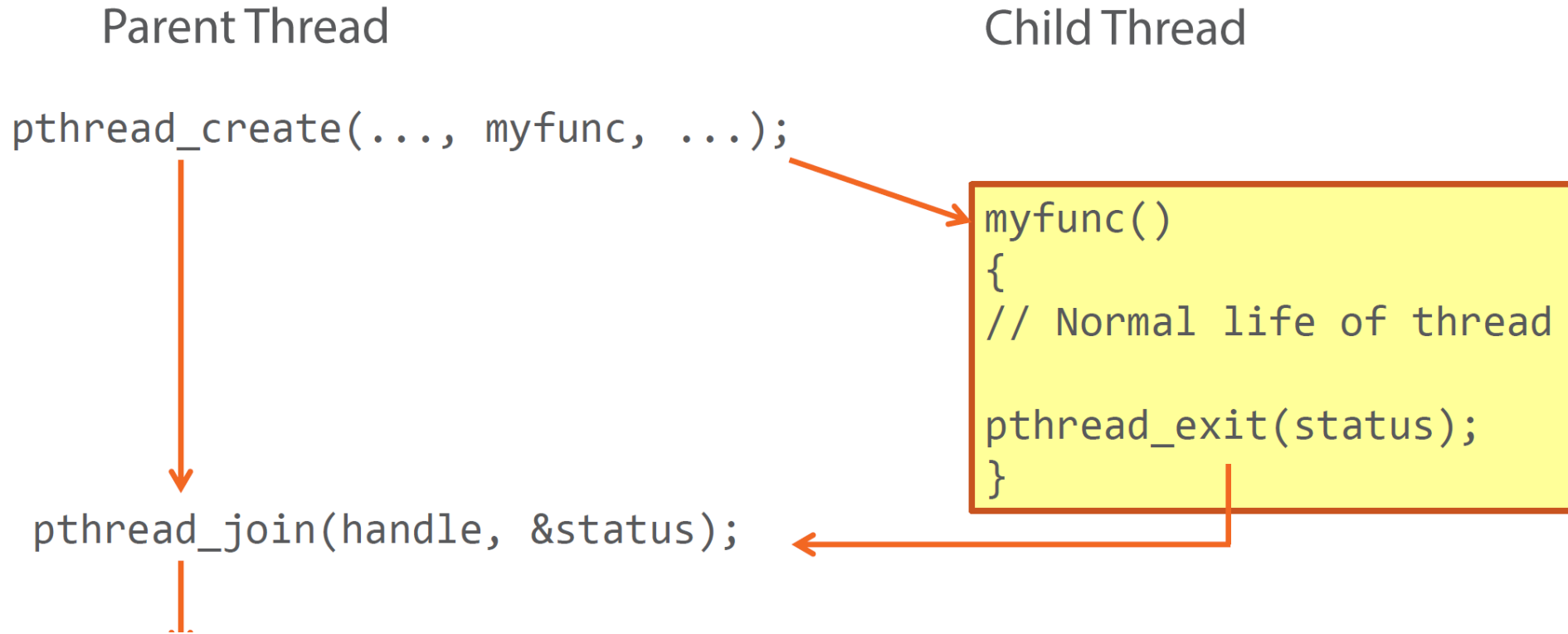
Thread Creation



Thread Termination

- A thread can terminate in several ways:
 1. By calling `pthread_exit(exit_status)`
 2. By returning from its top-level function
 3. By some other thread sending a cancellation request:
`pthread_cancel(handle);`
- Parent can wait for thread to finish and get exit status:
 - `pthread_join(handle, &exit_status);`
- Parent should detach thread if they don't need to join on it:
`pthread_detach(handle);`

Thread Life Cycle



Thread Example: thread1.c (NOT a perfect thread example)

```
void *thread_function(void *arg);

char message[] = "Hello World";

int main() {

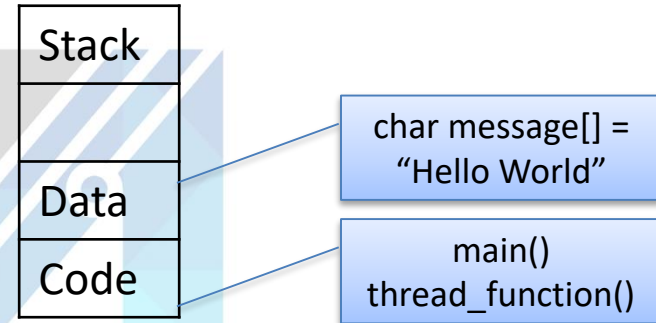
    res = pthread_create(&a_thread, NULL,
        thread_function, (void *)message);

    printf("Waiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);

    printf("Thread joined, it returned %s\n", (char
        *)thread_result);
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    int fd;
    printf("thread_function is running. Argument was
        %s\n", (char *)arg);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("Thank you for the CPU time");
}
```

thread1 process memory mapping:



Output:

Waiting for thread to finish...
thread_function is running. Argument was
Hello World
Thread joined, it returned **Thank you for
the CPU time**
Message is now Bye!


How pthread_create works?

fork() vs clone():

Unlike fork(), **clone() allows the child process to share** parts of its execution context with the calling process, such as the virtual address space, the table of file descriptors, and the table of signal handlers

One use of clone() is to implement threads:

multiple flows of control in a program that run concurrently in a shared address space.

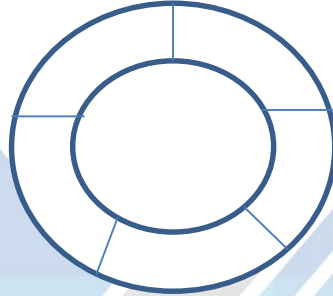


Producer Consumer Problem

(Perfect thread example)

Producer Consumer problem (thread2.c) - Perfect thread example

Producer and consumer threads are might not be thread safe functions.



```
char buffer[5];  
int counter = 0;
```

Producer() thread:

```
void producer(void)  
{  
    int in=0;  
    while(1)  
    {  
        while(counter == 5);  
        buffer[in] = 'A';  
        in = (in+1)%counter;  
        counter++;  
    }  
}
```

Producer Effected
whenever
buffer is FULL

Consumer() thread:

```
void consumer (void)  
{  
    int out=0;  
    char temp[10];  
    while(1)  
    {  
        while(counter == 0);  
        temp[out] = buffer[out];  
        out = (out+1)%counter;  
        counter--;  
    }  
}
```

Consumer Effected
whenever
buffer is EMPTY

Global variables (Pre-emptible) dangerous

Too many **variables** , if declared as **global**, then they remain in the memory till program execution is over.

Unprotected data : Data can be modified by any function.

Producer and Consumer thread both run at the same time counter++ and counter--, we can expect wrong output depending on the order.

Counter++

```
1. Reg1 = Counter;  
2. Reg1 += Reg1;  
3. Counter = Reg1;  
INR M; [M] <- [M] +1
```

Counter--

```
4. Reg2 = Counter;  
5. Reg2 -= Reg1;  
6. Counter = Reg2;  
DCR M; [M] <- [M] -1
```

assumption, Counter=4; After Counter++, counter--; counter= 4

1. Reg1=4
2. Reg1 = 4+1 (preemption)
4. Reg2= 4
5. Reg2 = 4-1
6. Counter = 3
3. **Counter =5**

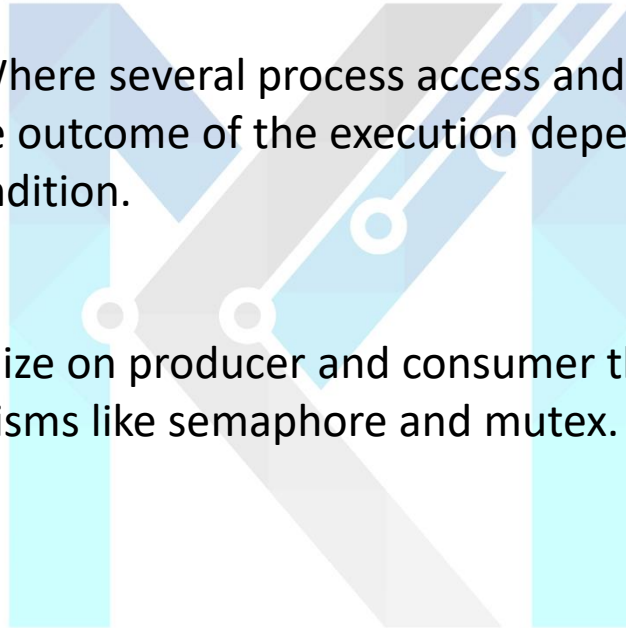
Producer consumer problem

Problem:

- Data Inconsistency
- Loss of Data
- Race around condition: Where several process access and manipulate the same data concurrently and the outcome of the execution depends on the particular order. It is called Race condition.

Solution:

1. Using signals to synchronize on producer and consumer threads.
2. Synchronization Mechanisms like semaphore and mutex.





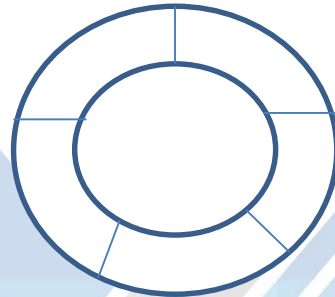
Thread Synchronization

Synchronizing Producer and Consumer threads using Signals

Producer Consumer problem using signals (threadsync signals.c)

Producer and consumer threads are now thread safe functions.

char buffer[5];



Producer() thread:

```
void producer(void)
{
    int in=0;
    while(1)
    {
        while(counter == 5);
        buffer[in] = 'A';
        in = (in+1)%counter;
        counter++;
    }
}
```

Consumer() thread:

```
void consumer (void)
{
    int out=0; char temp[10];
    while(1)
    {
        while(counter == 0);
        temp[out] = buffer[out];
        out = (out+1)%counter;
        counter--;
    }
}
```

Producer to Consumer

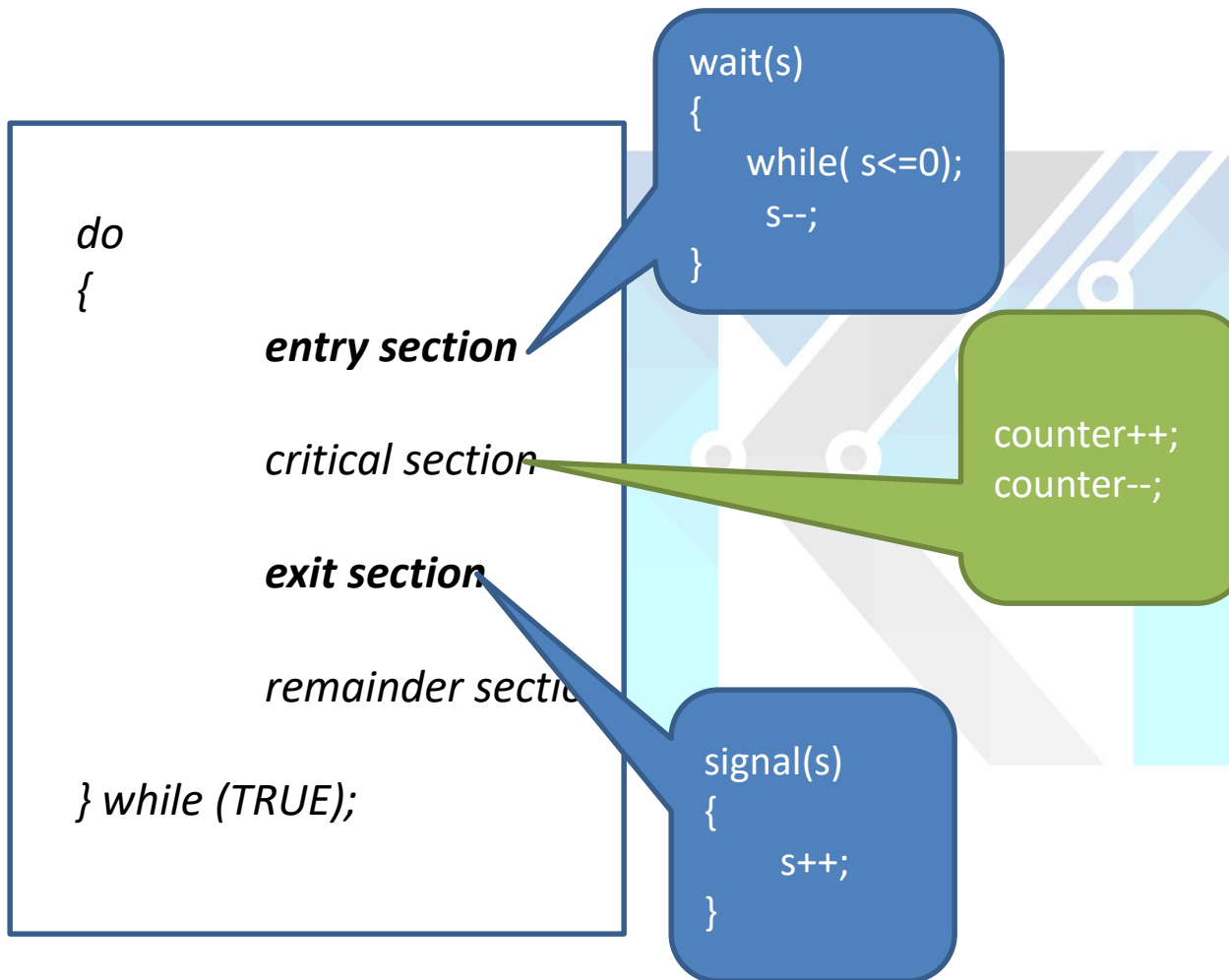
Consumer to Producer

A large, stylized, semi-transparent logo in the background consisting of the letters 'K' and 'M' in shades of blue and grey.

Thread Synchronization

Synchronizing Producer and Consumer threads using Semaphores

Synchronization Mechanisms



A semaphore **S** is an integer variable that, apart from initialization, is accessed only through two standard **atomic operations**:

The **wait()** operation was originally termed P.

signal () was originally called V.

Critical section: Critical section is that part of the program where shared resources are accessed.

POSIX Semaphores

sem_init(): initialize an unnamed semaphore.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

sem_post(): unlock a semaphore

```
int sem_post(sem_t *sem);
```

sem_wait(): lock a semaphore

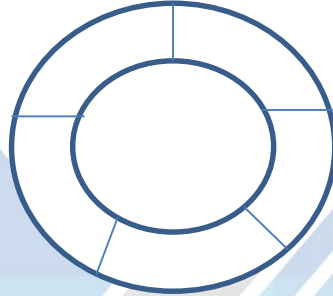
```
int sem_wait(sem_t *sem);
```

sem_destroy(): destroy an unnamed semaphore

```
int sem_destroy(sem_t *sem);
```


Producer Consumer problem using semaphores (threadsync semaphores.c)

Producer and consumer threads are now thread safe functions.



```
char buffer[5];  
//int counter = 0;  
sem_t empty=5,full=0;
```

Producer() thread:

```
void producer(void)  
{  
    int in=0;  
    while(1)  
    {  
        //while(counter == 5);  
        sem_wait (&empty);  
        buffer[in] = 'A';  
        in = (in+1)%counter;  
        // counter++;  
        sem_post (&full);  
    }  
}
```

Consumer() thread:

```
void consumer (void)  
{  
    int out=0; char temp[10];  
    while(1)  
    {  
        //while(counter == 0);  
        sem_wait (&full);  
        temp[out] = buffer[out];  
        out = (out+1)%counter;  
        // counter--;  
        sem_post(&empty);  
    }  
}
```