Concurrent Programming: Threads

CS 105: Computer Systems
Lecture 10

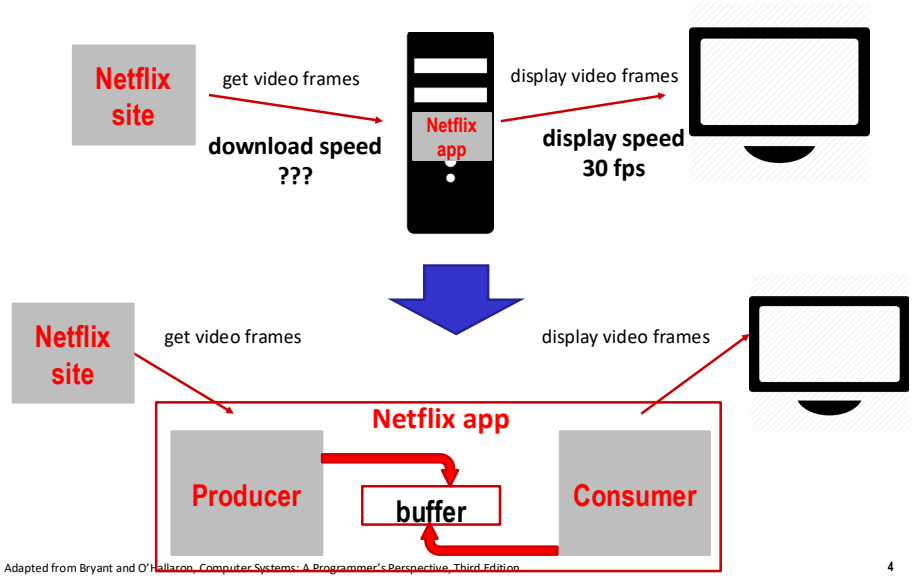
Melissa O'Neill

February 23, 2026

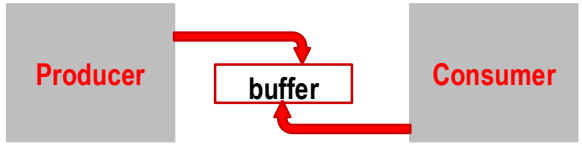
Learning Goals

- **Understand the memory model of *threads*, logical flows within a process**
- **Reason about what issues can arise in the correctness of a program when there are concurrent threads running**

Netflix: Example of producer-consumer



Producer-Consumer: how to implement?



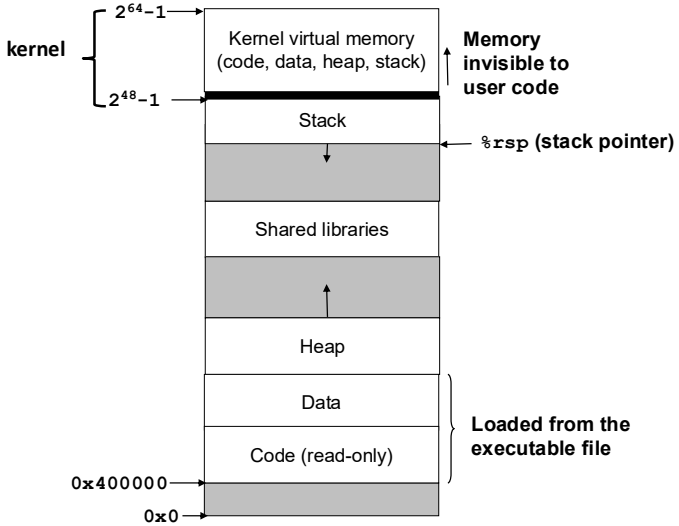
Option 1: Single process

```
While not done {
    Loop for a little while    // producer
        Get frames and store in buffer

    Loop for a little while    // consumer
        Display frames in buffer
}
```

Option 2: Separate processes

*Cumbersome to have a shared buffer… we'll look at **threads***

Recall: Memory layout of a process



Traditional View of a Process

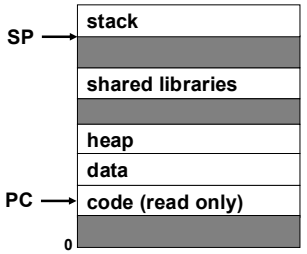
- Process = process context + code, data, and stack

Process context

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

Kernel context:
- VM structures
- File descriptor table
- …

Code, data, and stack



A Process With Multiple Threads

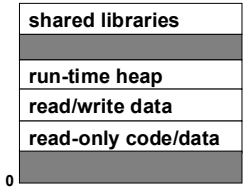
- Multiple threads can be associated with a process
  - Each thread has its *own* logical control flow (sequence of PC values)
  - Each thread *shares* the same code, data, and kernel context
  - Each thread has its own thread id (TID)

Thread 1 (main thread)

stack 1

Thread 1 context:
- Data registers
- Condition codes
- SP1
- PC1

Shared code and data



Kernel context:
- VM structures
- File descriptor table
- …

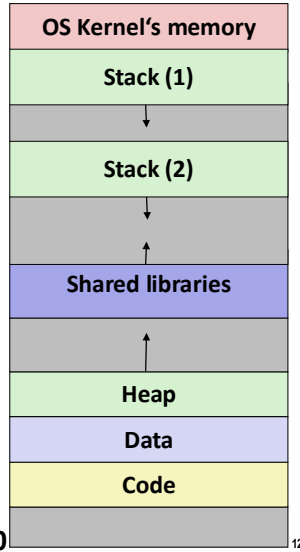
Thread 2 (peer thread)

stack 2

Thread 2 context:
- Data registers
- Condition codes
- SP2
- PC2

A Process with Multiple Threads

- Example: address space for a process with two threads
- Each thread has its own stack



Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently (maybe on different cores)
  - Each is context-switched

- **How threads and processes are different**
  - Threads share code and data, processes (typically) do not
  - Threads are somewhat "cheaper" than processes

Posix Threads (Pthreads) Interface

- ***Pthreads:*** **Standard interface for ~60 (!) functions that manipulate threads from C programs,** `<pthread.h>`
  - 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`

We'll come back to these for the Ring Buffer lab

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"  /* Err... */

void *threadfunc(void *vargp);

int main()
{
   pthread_t tid;
   Pthread_create(&tid, NULL, threadfunc, NULL);
   Pthread_join(tid, NULL);
   exit(0);
}
```

Thread ID

Thread attributes (usually NULL)

Thread routine

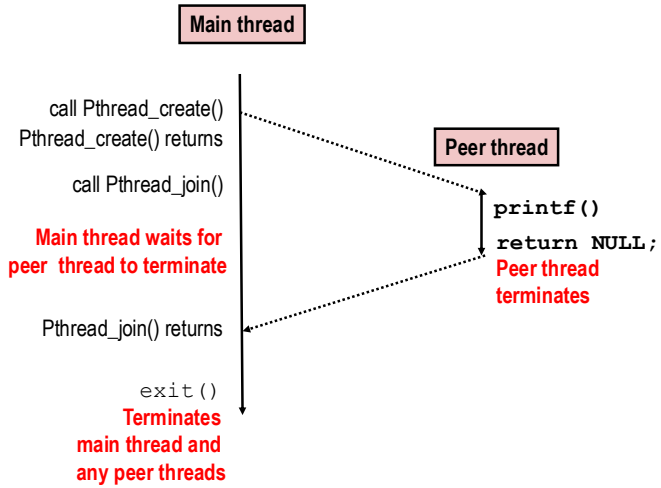
*Thread arguments Type is (void *p)*

*Return value Type is (void **p)*

p is a ptr to where the return value will be put, which will be a ptr

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

Execution of Threaded "hello, world"



Exercise: practice with creating threads

```
#include "csapp.h"
void *func(void *vargp);

int main() {
  pthread_t p1, p2;
  long *ret1, *ret2;

  Pthread_create(&p1, NULL, func, (void *) 100);  /* p1: func arg is 100 */
  Pthread_create(&p2, NULL, func, (void *) 200);  /* p2: func arg is 200 */

  Pthread_join(p1, (void **) &ret1);  /* p1: put return val at ret1 */
  Pthread_join(p2, (void **) &ret2);  /* p2: put return val at ret2 */
  printf("Thread results: %ld %ld\n", *ret1, *ret2);
  exit(0);
}
```

```
void *func(void *vargp) { /* thread routine */
  long input = (long) vargp;    /* cast the argument as a long */
  printf("%ld\n",input);  /* print the long and a newline */

  long *mynum = malloc(sizeof(long));          /* allocate a long on the heap */
  *mynum = 42 + input;                 /* dereference and set value */
  return mynum;                                          /* return heap address */
}
```


Exercise: practice with creating threads

Answer these questions using the code on the previous slide

1. **How many instances of the pointer variable `mynum` will there be? In which region of memory could `mynum` be?**

2. **What is a valid output from running this program?**

Threads Memory Model

- **Conceptual model:**
  - Multiple threads run within the context of a single process
  - Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, PC, condition codes, and general purpose registers
  - All threads share the remaining process context
    - Code, data, heap, and shared library segments of the process address space
    - Open files and handlers

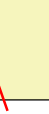
- **Operationally, this model is not strictly enforced:**
  - Register values are truly separate and protected, but…
  - Any thread can read and write the stack of any other thread

Definition: A variable x is shared if and only if multiple threads reference some instance of x.

Example Program to Illustrate Sharing

```
char **ptr;  /* global var */

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++) {
        Pthread_create(&tid,
            NULL,
            threadfunc,
            (void *)i);
    }
    Pthread_exit(NULL);
}
```

```
void* threadfunc(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);


    return NULL;
}
```



Peer threads reference main thread's stack indirectly through global `ptr` variable

Synchronizing Threads

- **Shared variables are handy...**

- **…but introduce the possibility of nasty *synchronization* errors.**

badcnt.c: Improper Synchronization

```
/* Global shared variable */
long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
   long num_iters;
   pthread_t tid1, tid2;

   num_iters = atoi(argv[1]);

   Pthread_create(&tid1, NULL,
      threadfunc, &num_iters);
   Pthread_create(&tid2, NULL,
      threadfunc, &num_iters);

   Pthread_join(tid1, NULL);
   Pthread_join(tid2, NULL);

   /* Check result */
   if (cnt != (2 * num_iters))
      printf("BOOM! cnt=%ld\n", cnt);
   else
      printf("OK cnt=%ld\n", cnt);
   exit(0);
}
```

```
/* Thread routine */
void *threadfunc(void *vargp)
{
  long j;
  long iters = *((long *)vargp);

  /* increment global var */
  for (j = 0; j < iters; ++j)
    ++cnt;

  return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

cnt should equal 20,000.

What went wrong?!

Assembly Code for Counter Loop

C code for counter loop in thread i

```
for (j = 0; j < iters; ++j)
    ++cnt;
```

Assembly code for thread i

```
    movq   (%rdi), %rcx
    testq  %rcx,%rcx
    jle    .L2
    movl   $0, %eax
.L3:
    movq   cnt(%rip),%rdx
    addq   $1, %rdx
    movq   %rdx, cnt(%rip)
    addq   $1, %rax
    cmpq   %rcx, %rax
    jne    .L3
.L2:
```

H_i : Head

L_i : Load `cnt` into %rdx

U_i : Update `cnt` in %rdx

S_i : Store `cnt` back to memory

T_i : Tail

`++cnt` is three lines of assembly!

Concurrent Execution

- ***Key idea:*** **In general, any sequentially consistent interleaving is possible, but some give an unexpected result!**
  - I_i denotes that thread i executes instruction I
  - $\%rdx_i$ is the content of %rdx in thread i's context (where `cnt` is put)

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

Thread 1's L,U,S (critical section)

Thread 2's L,U,S (critical section)

OK

Time

Exercise: Concurrent Execution

- **What is the value of cnt at the end given this ordering of interleaved instructions from the two threads?**

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1		-	
1	U_1		-	
2	H_2	-	-	
2	L_2	-		
1	S_1		-	
1	T_1		-	
2	U_2	-		
2	S_2	-		
2	T_2	-		

Time