UNM SCHOOL of ENGINEERING
Department of Computer Science

Concurrent Programming

CS 341: Intro. to Computer Architecture & Organization

Andree Jacobson

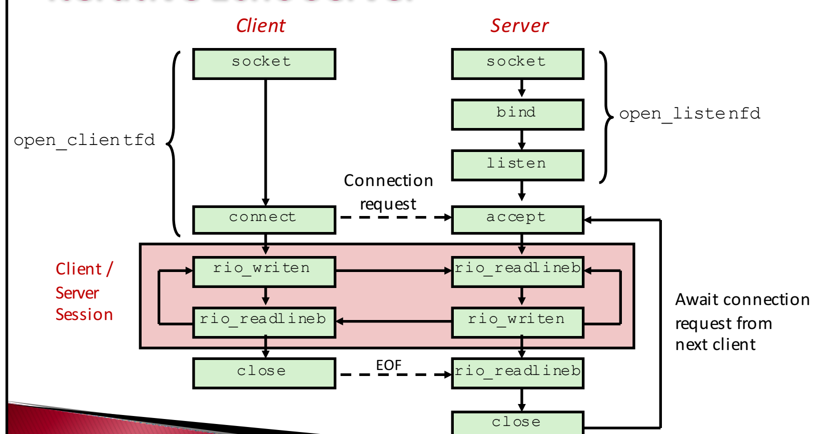
Concurrent Programming is Hard!

- The human mind tends to be (consciously) sequential
- The notion of time is often misleading
- We can't imagine complete sequence of computer events

Concurrent Programming is Hard!

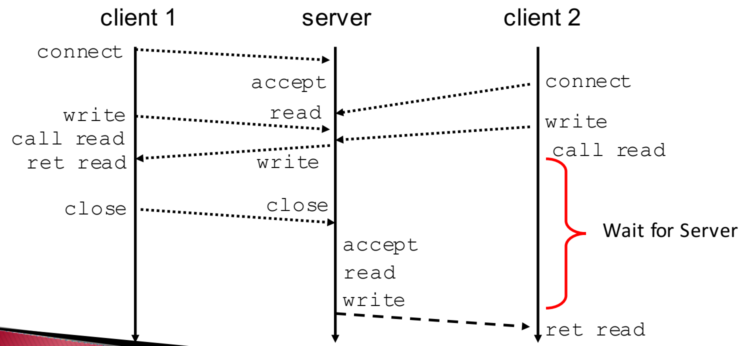
- Classical problem classes of concurrent programs:
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: musical chairs ☺
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Livelock / Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line

Iterative Echo Server



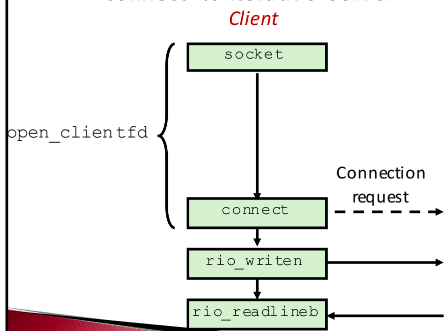
Iterative Servers

- Iterative servers process one request at a time



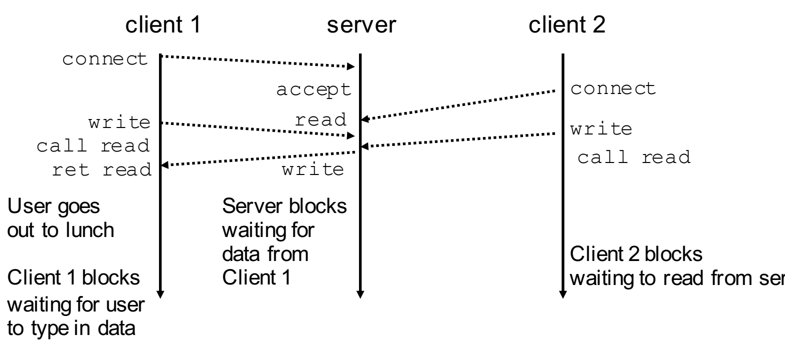
Where Does Second Client Block?

- Second client attempts to connect to iterative server



- Call to connect returns
  - Even though connection not yet accepted
  - Server side TCP manager queues request
  - Feature known as "TCP listen backlog"
- Call to rio_writen returns
  - Server side TCP manager buffers input data
- Call to rio_readlineb blocks
  - Server hasn't written anything for it to read yet.

Fundamental Flaw of Iterative Servers



- Solution: use *concurrent servers* instead
  - Concurrent servers use concurrent flows to serve concurrent clients

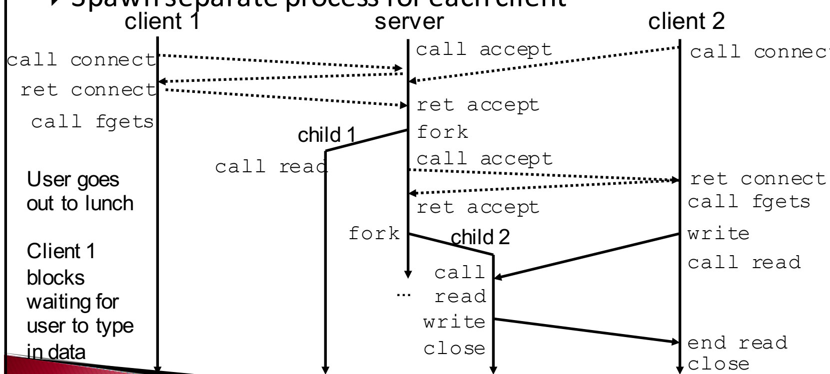
Creating Concurrent Flows

Allow server to handle multiple clients simultaneously

- 1. Processes
  - Kernel automatically interleaves multiple logical flows
  - Each flow has its own private address space
- 2. Threads
  - Kernel automatically interleaves multiple logical flows
  - Each flow shares the same address space
- 3. I/O multiplexing with `select()`
  - Programmer manually interleaves multiple logical flows
  - All flows share the same address space
  - Relies on lower-level system abstractions

Concurrent Servers: Multiple Processes

- Spawn separate process for each client



Review: Iterative Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);

    listenfd = Open_listenfd(port);
    while (1) {
      connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
      echo(connfd);
      Close(connfd);
    }
    exit(0);
}
```

- Accept a connection request
- Handle echo requests until client terminates

Process-Based Concurrent Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen=sizeof(clientaddr);

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(port);
    while (1) {
    connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    if (Fork() == 0) {
        Close(listenfd); /* Child closes its listening socket */
        echo(connfd);    /* Child services client */
        Close(connfd);   /* Child closes connection with client */
        exit(0);         /* Child exits */
    }
    Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

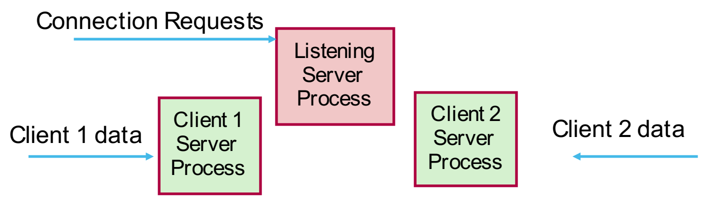
Fork separate process for each client
Does not allow any communication between different client handlers

Process-Based Concurrent Server (cont)

```
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
    ;
    return;
}
```

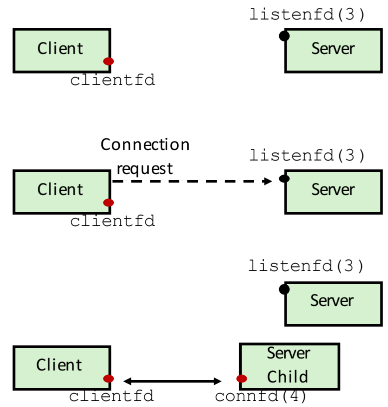
- Reap all zombie children

Process Execution Model



- Each client handled by independent process
- No shared state between them
- Both parent & child have copies of listenfd and connfd
  - Parent must close connfd
  - Child must close listenfd

Concurrent Server: `accept` Illustrated



1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

2. Client makes connection request by calling and blocking in `connect`

3. Server returns `connfd` from `accept`. Forks child to handle client. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

Implementation Must-dos With Process-Based Designs

- Listening server process must reap zombie children
  - to avoid fatal memory leaks
- Listening server process must `close` its copy of `connfd`
  - Kernel keeps reference for each socket/open file
  - After fork, `refcnt(connfd) = 2`
  - Connection will not be closed until `refcnt(connfd) == 0`

Pros and Cons of Process-Based Designs

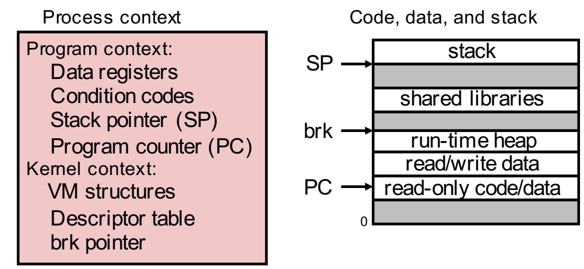
- + Handle multiple connections concurrently
- + Clean sharing model
  - descriptors (no)
  - file tables (yes)
  - global variables (no)
- + Simple and straightforward
- – Additional overhead for process control
- – Nontrivial to share data between processes
  - Requires IPC (interprocess communication) mechanisms
    - FIFO's (named pipes), System V shared memory and semaphores

Approach #2: Multiple Threads

- Very similar to approach #1 (multiple processes)
  - but, with threads instead of processes

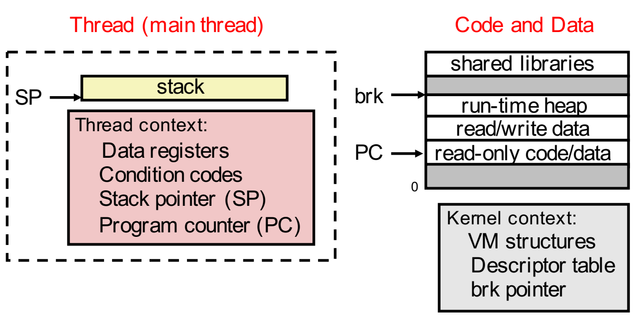
Traditional View of a Process

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



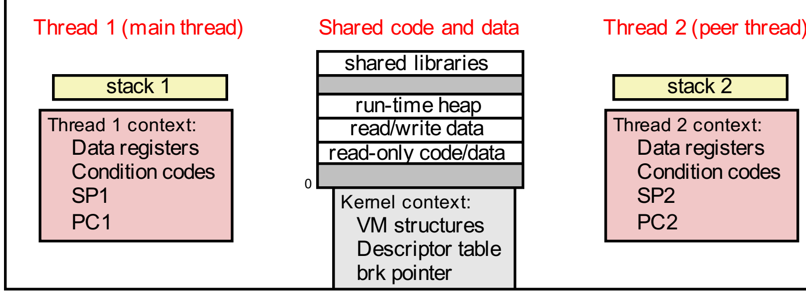
Alternate View of a Process

- Process = thread + code, data, and kernel context



A Process With Multiple Threads

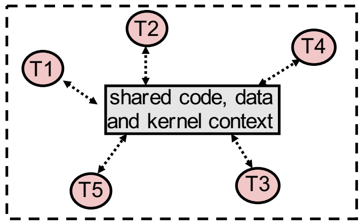
- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
    - Share common virtual address space (inc. stacks)
  - Each thread has its own thread id (TID)



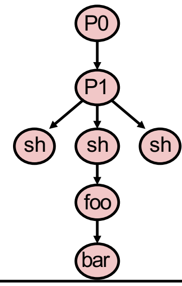
Logical View of Threads

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

Threads associated with a process

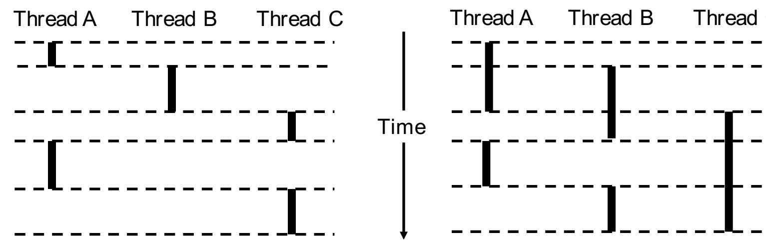


Process hierarchy



Thread Execution

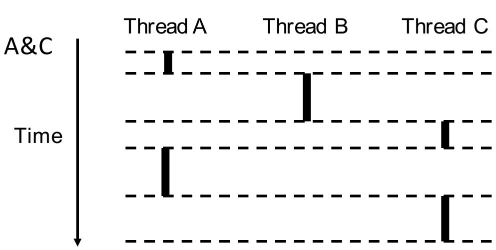
- Single Core Processor
  - Simulate concurrency by time slicing
- Multi-Core Processor
  - Can have true concurrency



Run 3 threads on 2 cores

Logical Concurrency

- Two threads are (logically) concurrent if their flows overlap in time; otherwise, they are sequential
- Examples:
  - Concurrent: A & B, A&C
  - Sequential: B & C



Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- How threads and processes are different
  - Threads share code and some data
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) is about twice as expensive as thread control

Posix Threads (Pthreads) Interface

- *Pthreads:* Standard interface for ~60 functions that manipulate threads from C programs
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [terminates all threads] , `RET` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`
    - `pthread_cond_init`
    - `pthread_cond_[timed]wait`

The Pthreads "hello, world" Program

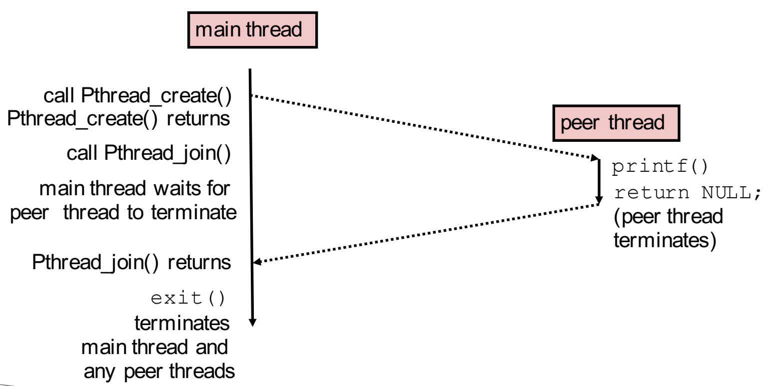
```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

Thread attributes (usually NULL)

*Thread arguments (void *p)*

*return value (void **p)*

Execution of Threaded"hello, world"



Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv) {
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen=sizeof(clientaddr);
    pthread_t tid;

    int listenfd = Open_listenfd(port);
    while (1) {
      int *connfdp = Malloc(sizeof(int));
      *connfdp = Accept(listenfd,
                        (SA *) &clientaddr, &clientlen);
      Pthread_create(&tid, NULL, echo_thread, connfdp);
    }
}
```

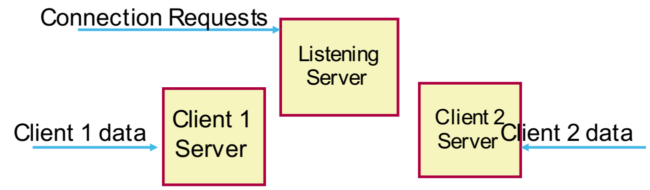
- Spawn new thread for each client
- Pass it copy of connection file descriptor
- Note use of Malloc()!
  - Without corresponding Free()

Thread-Based Concurrent Server (cont)

```
/* thread routine */
void *echo_thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
```

- Run thread in "detached" mode
  - Runs independently of other threads
  - Reaped when it terminates
- Free storage allocated to hold clientfd
  - "Producer-Consumer" model

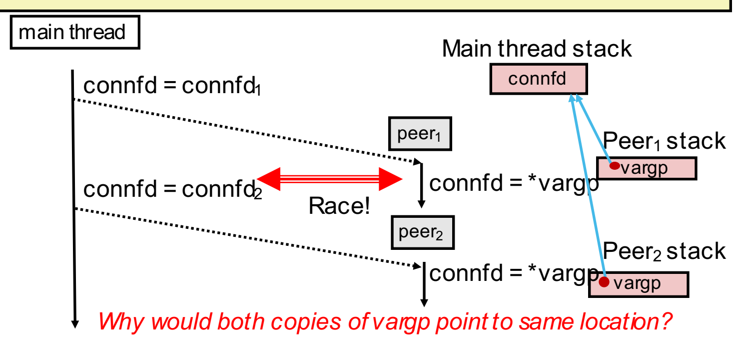
Threaded Execution Model



- Multiple threads within single process
- Some state between them
  - File descriptors

Potential Form of Unintended Sharing

```
    while (1) {
      int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
      Pthread_create(&tid, NULL, echo_thread, (void *) &connfd);
    }
}
```



Why would both copies of vargp point to same location?

Could this race occur?

Main

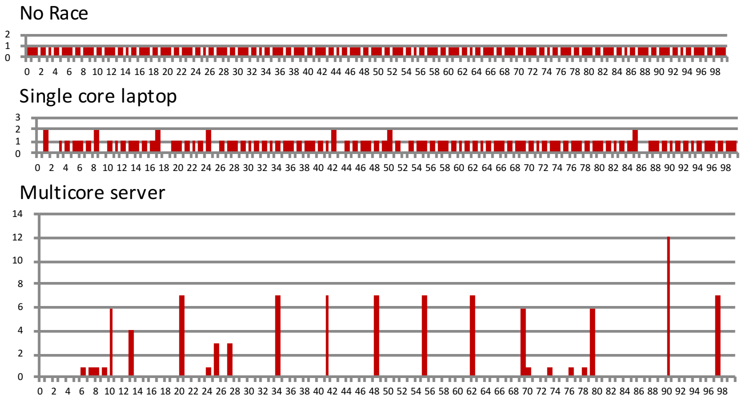
```
int i;
for (i = 0; i < 100; i++) {
  Pthread_create(&tid, NULL,
                 thread, &i);
}
```

Thread

```
void *thread(void *vargp)
{
  int i = *((int *)vargp);
  Pthread_detach(pthread_self());
  save_value(i);
  return NULL;
}
```

- Race Test
  - If no race, then each thread would get different value of i
  - Set of saved values would consist of one copy each of 0 through 99.

Experimental Results



- The race can really happen!

Issues With Thread-Based Servers

- 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.
  - Default state is joinable.
    - use `pthread_detach(pthread_self())` to make detached.
- Must be careful to avoid unintended sharing.
  - For example, passing pointer to main thread's stack
    `Pthread_create(&tid, NULL, thread, (void *)&connfd);`
- All functions called by a thread must be *thread-safe*

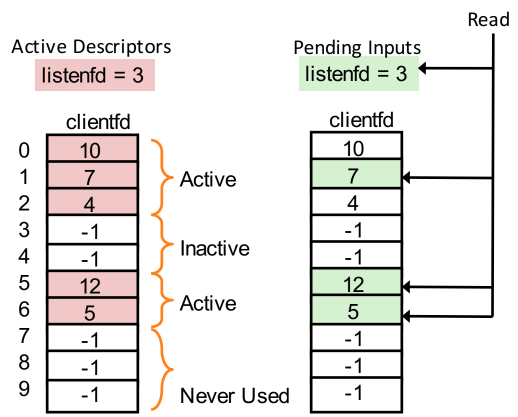
Pros and Cons of Thread-Based Designs

- + Easy to share data structures between threads
  - e.g., logging information, file cache.
- + Threads are more efficient than processes.
- **–** Unintentional sharing can lead to subtle, hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads.
  - Hard to know which data shared & which private
  - Hard to detect by testing
    - Probability of bad race outcome very low
    - But nonzero!

Event-Based Concurrent Servers Using I/O Multiplexing

- Use library functions to construct scheduler within single process
- Server maintains set of active connections
  - Array of connfd's
- Repeat:
  - Determine which connections have pending inputs
  - If listenfd has input, then accept connection
    - Add new connfd to array
  - Service all connfd's with pending inputs
- Details in book

I/O Multiplexed Event Processing



Pros and Cons of I/O Multiplexing

- + One logical control flow.
- + Can single-step with a debugger.
- + No process or thread control overhead.
  - Design of choice for high-performance Web servers and search engines.
- – Significantly more complex to code than process- or thread-based designs.
- – Hard to provide fine-grained concurrency
  - E.g., our example will hang up with partial lines.
- – Cannot take advantage of multi-core
  - Single thread of control

Approaches to Concurrency

- Processes
  - Hard to share resources: Easy to avoid unintended sharing
  - High overhead in adding/removing clients
- Threads
  - Easy to share resources: Perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug
    - Event orderings not repeatable
- I/O Multiplexing
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core

UNM SCHOOL of ENGINEERING
Department of Computer Science

Synchronization

CS 341: Intro. to Computer Architecture & Organization

Prof. Dorian Arnold

Why we need Synchronization

- Concurrent access to shared data can lead to inconsistencies
- Need synchronization amongst sharers to guarantee consistency

Producer/Consumer Example

```
while (true) {
    while (count == BUFFER_SIZE)
        ; // do nothing

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

Producer Code

Producer/Consumer Example

```
while (true)  {
    while (count == 0)
    ; // do nothing

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    /* consume the item nextConsumed */
}
```

Consumer Code

The Critical Section Problem

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

Concurrency Terminology

- Race condition exists when output varies depending on access (write) order
- Critical section: section of code used by multiple processes (or threads) to update shared data
  - Entry section: request access to critical section
  - Exit section: notify departure from critical section
  - Remainder section: code outside critical section
- lock: abstraction/data structure used to access code in critical section

Critical Sections

```
do {
    entry section
        critical section
    exit section
        remainder section
} while( condition )
```


Solutions to Critical Section Problem

- Mutual exclusion: only one process can be executing code in a critical section at any point in time
- Progress: if no process is in critical section, process(es) wishing to enter the critical section must be allowed to do so
- Bounded wait: a process cannot be permanently blocked from entering a critical section

Enforcing Mutual Exclusion

- *Question:* How can we guarantee a safe trajectory?
- Answer: We must ***synchronize*** the execution of the threads so that they never have an unsafe trajectory.
  - i.e., need to guarantee ***mutually exclusive access*** to critical regions
- Classic solution:
  - Semaphores (Edsger Dijkstra)
- Other approaches (out of our scope)
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)

Semaphores Operations

binary semaphore (mutex lock)

counting semaphore

spinlock

```
//initialization
S = 1;
S = N;

wait(S)
{
    while ( S <= 0 ) ;
    S--;
}

signal( S )
{
    S++;
}
```

When might spinlocks be useful?

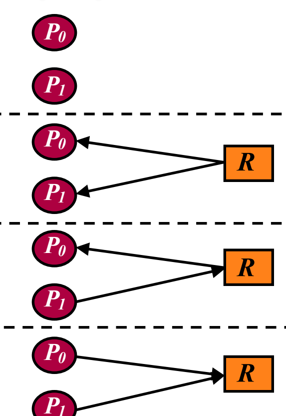
Entry and Exit using Semaphores

```
S = 1;

do {
    wait(S);

    // critical section

    signal(S);

    // remainder section

} while (TRUE);
```


Non-spin-lock Semaphores

```
typedef struct {
    int value;
    struct process *list;
} semaphore;

wait( semaphore *S ) {
    S->value--;
    if (S->value < 0) {
        S->list->add( current_process );
        block();
    }
}

signal( semaphore *S ) {
    S->value++;
    if (S->value <= 0) {
        S->list->remove( selected_process );
        wakeup( selected_process );
    }
}
```


Why Synchronization (Summary)



No sharing?

No synchronization needed

Read-only sharing?

No synchronization needed

Read-write sharing?

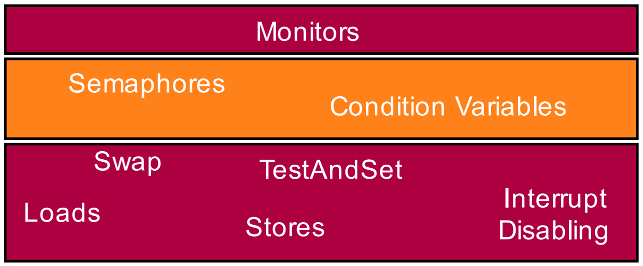
Synchronization potentially needed

Write-write sharing?

Synchronization needed

Synchronization Mechanisms

- Guarantee mutual exclusion when/where necessary
  - Higher-level primitives for programming convenience



Deadlock

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
      P0                    P1
wait (S);             wait (Q);
wait (Q);             wait (S);
  .                     .
  .                     .
  .                     .
signal (S);           signal (Q);
signal (Q);           signal (S);
```

54

Synchronization Issues

- Subtle synchronization errors
- Races
- Priority Inversion
- Deadlock