CS 33

Multithreaded Programming II

Example (1)

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

#define M   3
#define N   4
#define P   5

long A[M][N];
long B[N][P];
long C[M][P];

void *matmult(void *);
```

```
main( ) {
  long i;
  pthread_t thr[M];
  int error;

  // initialize the matrices
  ...
```

Example (2)

```
  for (i=0; i<M; i++) {   // create worker threads
   if (error = pthread_create(
       &thr[i],
       0,
       matmult,
       (void *)i)) {
     fprintf(stderr, "pthread_create: %s", strerror(error));
     exit(1);
   }
  }
  for (i=0; i<M; i++) // wait for workers to finish their jobs
   pthread_join(thr[i], 0)
  /* print the results  ... */
}
```

Example (3)

```
void *matmult(void *arg) {
  long row = (long)arg;
  long col;
  long i;
  long t;

  for (col=0; col < P; col++) {
   t = 0;
   for (i=0; i<N; i++)
     t += A[row][i] * B[i][col];
   C[row][col] = t;
  }
  return(0);
}
```

Compiling It

```
% gcc –o mat mat.c -pthread
```

Termination

```
pthread_exit((void *) value);

return((void *) value);


pthread_join(thread, (void **) &value);
```

Detached Threads

```
start_servers( ) {
  pthread_t thread;
  int i;

  for (i=0; i<nr_of_server_threads; i++) {
    pthread_create(&thread, 0, server, 0);
    pthread_detach(thread);
  }
  ...
}

void *server(void * arg ) {
  ...
}
```

Worker Threads

```
int main() {
  pthread_t thread[10];
  for (int i=0; i<10; i++)
    pthread_create(&thread[i], 0,
        worker, (void *)i);
  return 0;
}

void *worker(...) {...}
```

Termination

```
pthread_exit((void *) value);

return((void *) value);


pthread_join(thread, (void **) &value);

exit(code);  // terminates process!
```

Complications

```
void relay(int left, int right) {
  pthread_t LRthread, RLthread;

  pthread_create(&LRthread,
      0,
      copy,
      left, right);        // Can't do this ...
  pthread_create(&RLthread,
      0,
      copy,
      right, left);        // Can't do this  ...
}
```

Multiple Arguments

```
typedef struct args {
  int src;
  int dest;
} args_t;

void relay(int left, int right) {
  args_t LRargs, RLargs;
  pthread_t LRthread, RLthread;
  ...
  pthread_create(&LRthread, 0, copy, &LRargs);
  pthread_create(&RLthread, 0, copy, &RLargs);
}
```

Multiple Arguments

Quiz 1

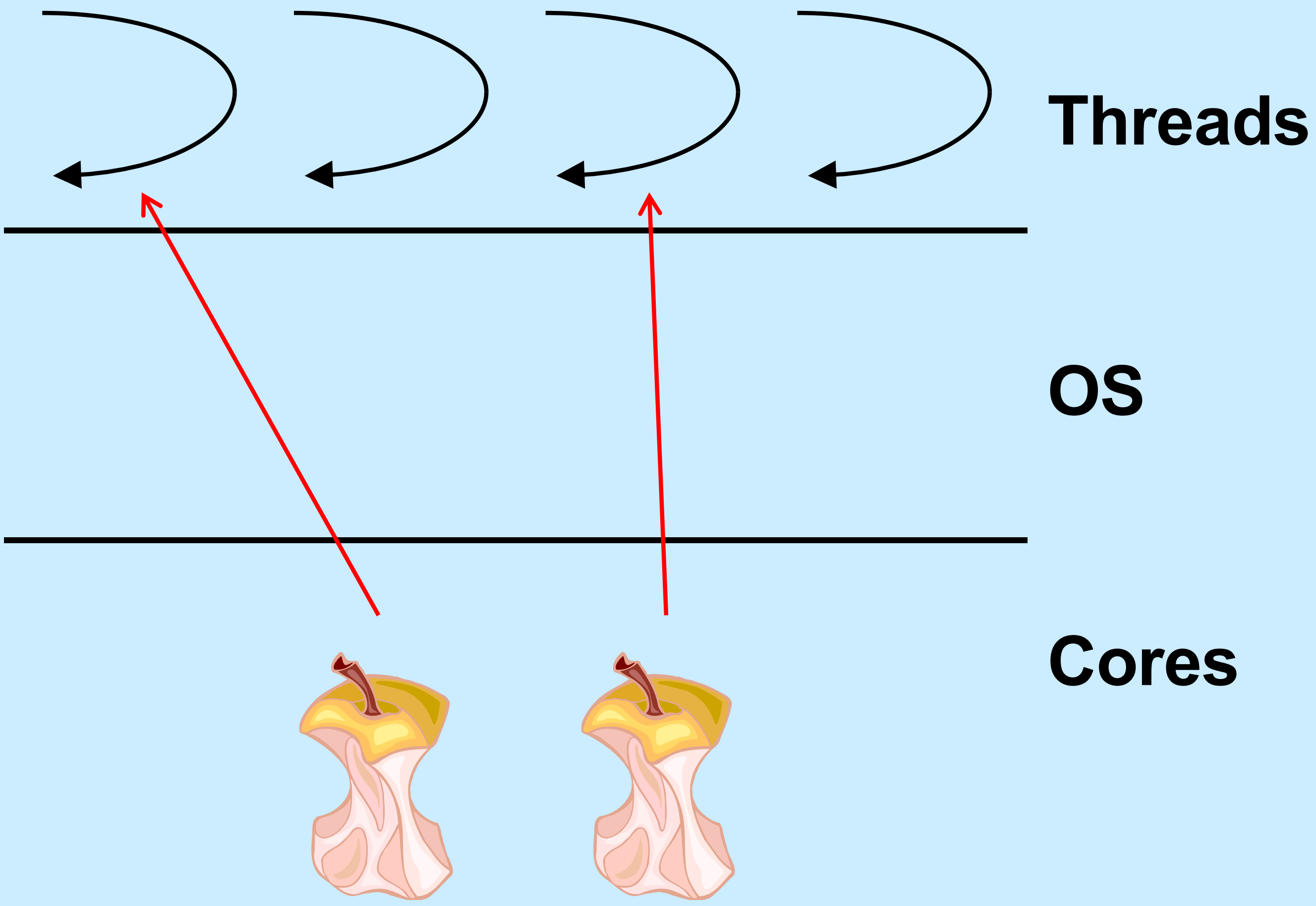
Does this work?

a) yes

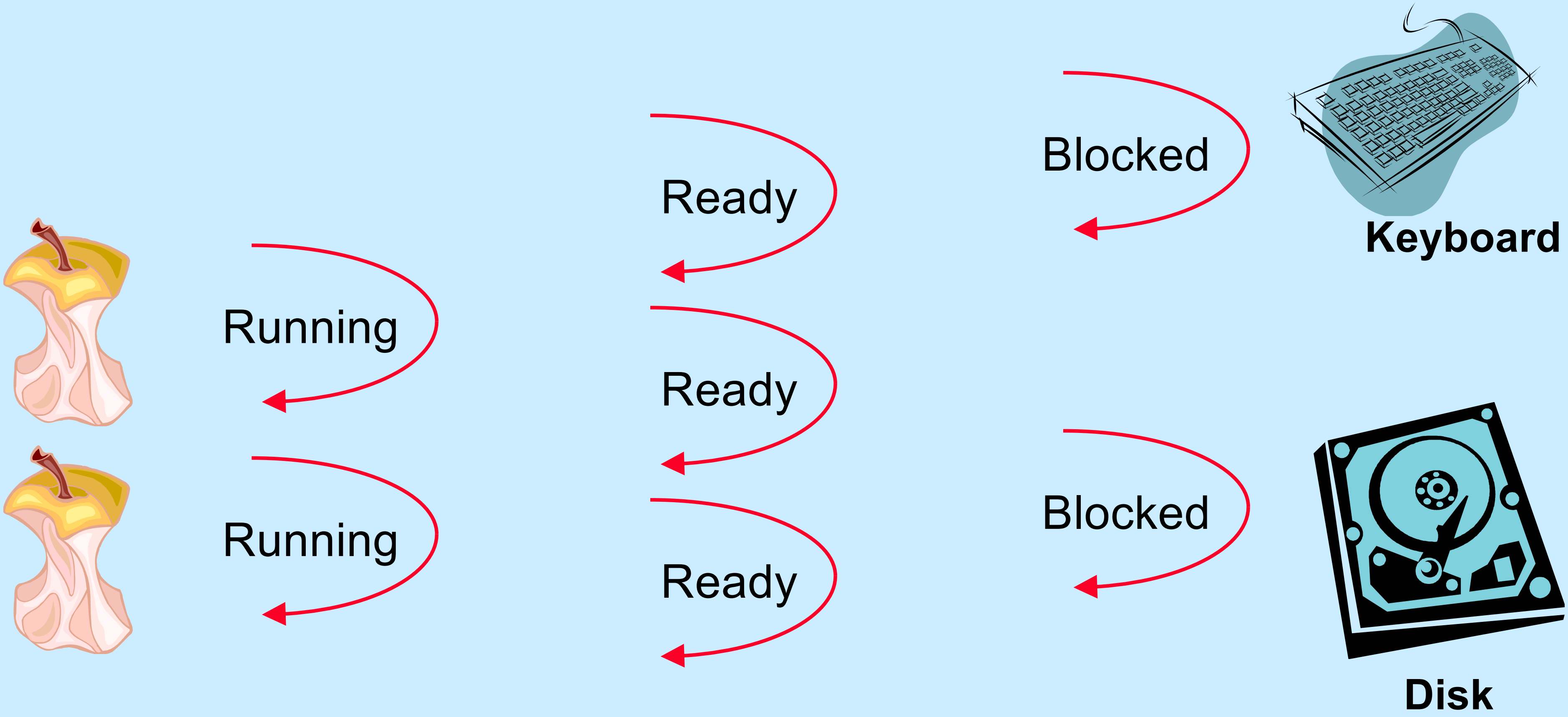
b) no

```
typedef struct args {
  int src;
  int dest;
} args_t;

void relay(int left, int right) {
  args_t LRargs, RLargs;
  pthread_t LRthread, RLthread;
  ...
  pthread_create(&LRthread, 0, copy, &LRargs);
  pthread_create(&RLthread, 0, copy, &RLargs);
}
```

Execution



Multiplexing Processors



Blocked

Keyboard

Ready

Running

Ready

Blocked

Running

Ready

Disk

Quiz 2

```
  pthread_create(&tid, 0, tproc, (void *)1);
  pthread_create(&tid, 0, tproc, (void *)2);

  printf("T0\n");

  ...

void *tproc(void *arg) {
  printf("T%dl\n", (long)arg);
  return 0;
}
```

In which order are things printed?

a) T0, T1, T2

b) T1, T2, T0

c) T2, T1, T0

d) indeterminate

Cost of Threads

```
int main(int argc, char *argv[]) {
   ...
   val = niters/nthreads;

   for (i=0; i<nthreads; i++)
      pthread_create(&thread, 0, work, (void *)val);
   pthread_exit(0);
   return 0;
}

void *work(void *arg) {
   long n = (long)arg; int i, j; volatile long x;

   for (i=0; i<n; i++) {
      x = 0;
      for (j=0; j<1000; j++)
         x = x*j;
   }
   return 0;
}
```

Cost of Threads

```
int main(int argc, char *argv[]) {
   ...
   val = niters/nthreads;

   for (i=0; i<nthreads; i++)
      pthread_create(&thread, 0, work, (void *)val);
   pthread_exit(0);
   return 0;
}

void *work(void *arg) {
   long n = (long)arg; int i, j; volatile long x;

   for (i=0; i<n; i++) {
      x = 0;
      for (j=0; j<1000; j++)
         x = x*j;
   }
   return (void *)x;
}
```

Not a Quiz

This code runs in time *n* on a 4-core processor when *nthreads* is 8. It runs in time *p* on the same processor when *nthreads* is 400.

a) *n* << *p* (slower)
b) *n* ≈ *p* (same speed)
c) *n* >> *p* (faster)

Problem

```
  pthread_create(&thread, 0, start, 0);

  …


void *start(void *arg) {
  long BigArray[128*1024*1024];
  …
  return 0;
}
```

Thread Attributes

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);

...

/* establish some attributes */

...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
```

Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 130*1024*1024);

...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
```

Mutual Exclusion

Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
 /*
   movl x,%eax
   incr %eax
   movl %eax,x
 */
```

Thread 2:

```
x = x+1;
 /*
   movl x,%eax
   incr %eax
   movl %eax,x
 */
```

Quiz 3

Suppose gcc produces the following code. Will it still be the case that x’s value might not be incremented by 2?

- a) yes
- b) no

Thread 1:

```
x = x+1;
 /*
   incr x
  */
```

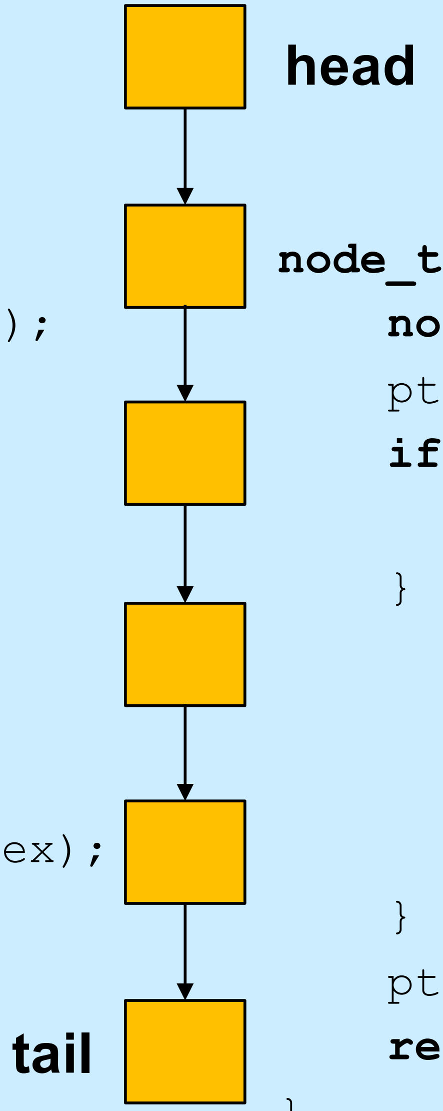
Thread 2:

```
x = x+1;
 /*
   incr x
  */
```

POSIX Threads Mutual Exclusion

```
pthread_mutex_t m =
     PTHREAD_MUTEX_INITIALIZER;
     // shared by both threads
int x; // ditto

  pthread_mutex_lock(&m);

  x = x+1;

  pthread_mutex_unlock(&m);
```

A Queue



```
void enqueue(node_t *item) {
    pthread_mutex_lock(&mutex);
    item->next = NULL;
    if (tail == NULL) {
        head = item;
    } else {
        tail->next = item;
    }
    tail = item;
    pthread_mutex_unlock(&mutex);
}
```

```
node_t *dequeue() {
    node_t *ret;
    pthread_mutex_lock(&mutex);
    if (head == NULL) {
        ret = NULL;
    } else {
        ret = head;
        head = head->next;
        if (head == NULL)
            tail = NULL;
    }
    pthread_mutex_unlock(&mutex);
    return ret;
}
```

Correct Usage

```
pthread_mutex_lock(&m);

// critical section

pthread_mutex_unlock(&m);
```

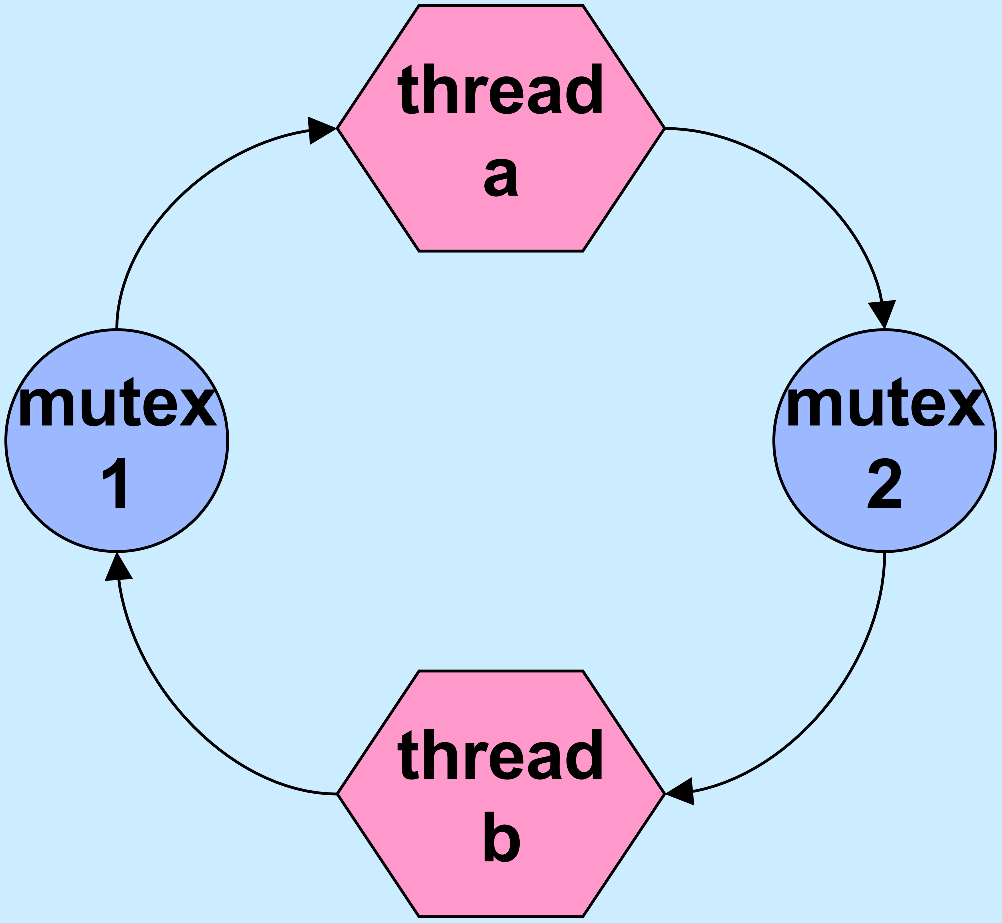
```
...
// in thread 1
pthread_mutex_lock(&m);

// critical section

return;


...
// in thread 2
pthread_mutex_unlock(&m);
```

Taking Multiple Locks

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
  pthread_mutex_lock(&m2);
  /* use object 2 */
  pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

Preventing Deadlock



Taking Multiple Locks, Safely

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

Practical Issues with Mutexes

- Used a lot in multithreaded programs
  - speed is really important
    - shouldn't slow things down much in the success case
  - checking for errors slows things down (a lot)
    - thus errors aren't checked by default

Set Up

```
int pthread_mutex_init(pthread_mutex_t *mutexp,
    pthread_mutexattr_t *attrp)

int pthread_mutex_destroy(pthread_mutex_t *mutexp)

int pthread_mutexattr_init(pthread_mutexattr_t *attrp)

int pthread_mutexattr_destroy(pthread_mutexattr_t *attrp)
```

Stupid (i.e., Common) Mistakes …

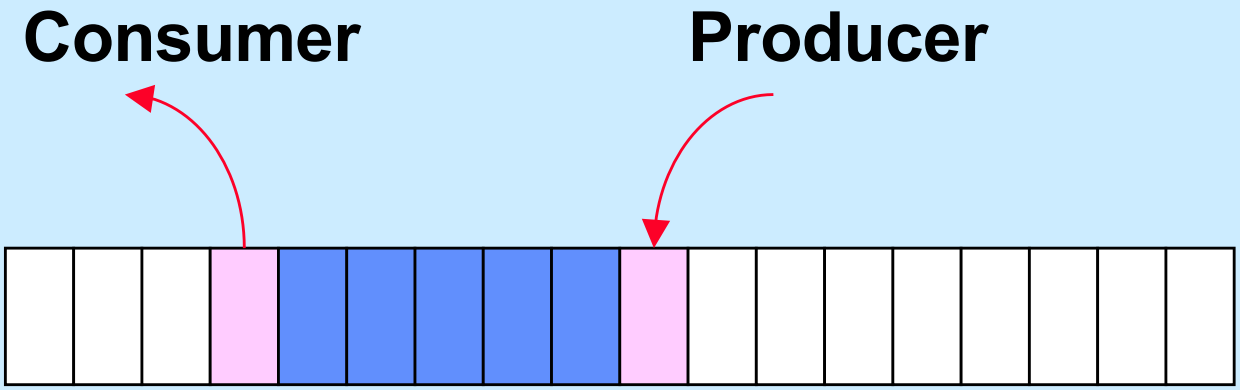
```
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m1);
  // really meant to lock m2 ...
```

```
pthread_mutex_lock(&m1);
  ...
pthread_mutex_unlock(&m2);
  // really meant to unlock m1 ...
```

Runtime Error Checking

```
pthread_mutexattr_t err_chk_attr;
pthread_mutexattr_init(&err_chk_attr);
pthread_mutexattr_settype(&err_chk_attr,
      PTHREAD_MUTEX_ERRORCHECK);

pthread_mutex_t mut1;
pthread_mutex_init(&mut1, &err_chk_attr);

pthread_mutex_lock(&mut1);

if (pthread_mutex_lock(&mut1) == EDEADLK)
  fprintf(stderr, "error caught at runtime\n");

if (pthread_mutex_unlock(&mut2) == EPERM)
  fprintf(stderr, "another error: you didn’t lock it!\n");
```

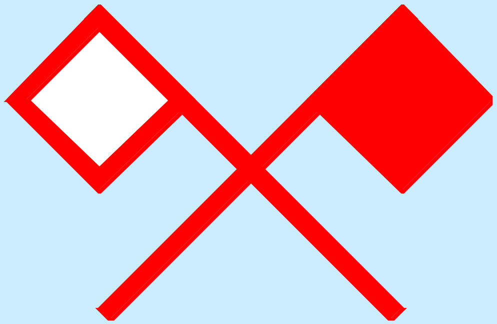
Producer-Consumer Problem



Guarded Commands

```
when (guard) [
  /*
    once the guard is true, execute this
    code atomically
   */

  ...

]
```

Semaphores



- **P(S) operation:**

```
when (S > 0) [
   S = S - 1;
]
```

- **V(S) operation:**

```
[S = S + 1;]
```

Quiz 4

```
semaphore S = 1;
int count = 0;

void proc( ) {
   P(S);
   count++;
   ...
   count--;
   V(S);
}
```

The function proc is called concurrently by n threads. What's the maximum value that count will take on?

a) 1
b) 2
c) n
d) indeterminate

- **P(S) operation:**

```
      when (S > 0) [
         S = S - 1;
      ]
```

- **V(S) operation:**

```
      [S = S + 1;]
```

Producer/Consumer with Semaphores

```
Semaphore empty = BSIZE;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  if (++nextin >= BSIZE)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  if (++nextout >= BSIZE)
    nextout = 0;
  V(empty);
  return item;
}
```

POSIX Semaphores

```
#include <semaphore.h>

int sem_init(sem_t *semaphore, int pshared, int init);
int sem_destroy(sem_t *semaphore);
int sem_wait(sem_t *semaphore);
    /* P operation */
int sem_trywait(sem_t *semaphore);
    /* conditional P operation */
int sem_post(sem_t *semaphore);
    /* V operation */
```

Producer-Consumer with POSIX Semaphores

```
sem_init(&empty, 0, BSIZE);
sem_init(&occupied, 0, 0);
int nextin = 0;
int nextout = 0;
```

```
void produce(char item) {

  sem_wait(&empty);
  buf[nextin] = item;
  if (++nextin >= BSIZE)
   nextin = 0;
  sem_post(&occupied);
}
```

```
char consume( ) {
  char item;
  sem_wait(&occupied);
  item = buf[nextout];
  if (++nextout >= BSIZE)
   nextout = 0;
  sem_post(&empty);
  return item;
}
```