# CSE 306 Operating Systems

## Concurrency: Mutual Exclusion and Synchronization

YoungMin Kwon

# Semaphores

- Fundamental Principles
  - Two or more processes can cooperate by means of a simple signal
    - A process can be forced to stop at a specific place
    - Resume execution on receiving a signal
  - Any complex coordination requirement can be satisfied

# Semaphore

- `semSignal(s)`
  - To transmit a signal via semaphore s
- `semWait(s)`
  - To receive a signal via semaphore s
  - If no corresponding signal has been sent, the process will be suspended until the signal is sent

# Semaphores

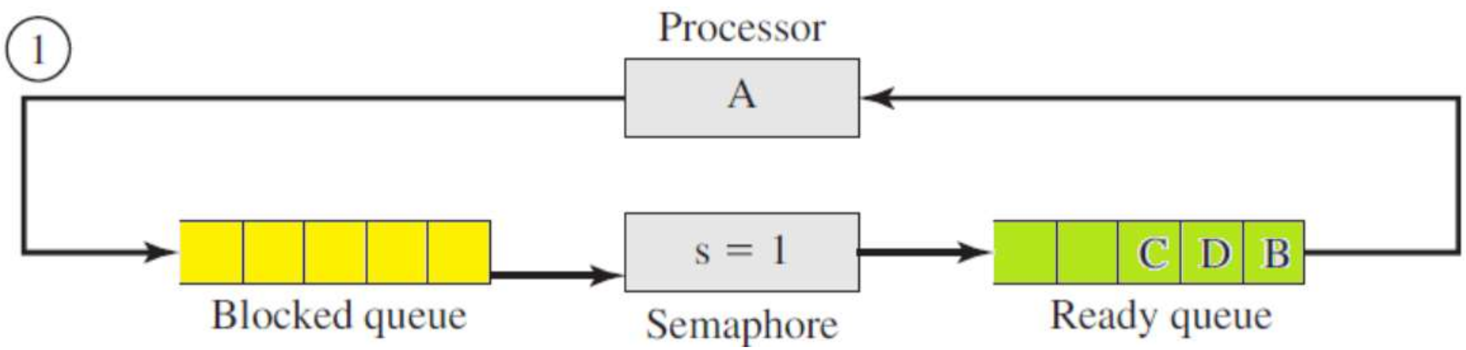
- Semaphore operations
  - Initialization: a semaphore may be initialized to a nonnegative integer value
  - semWait: decrements the semaphore value
    - If the value becomes negative, the process will be blocked
    - Otherwise, the process continues its execution
  - semSignal: increments the semaphore value
    - If the resulting value is non-positive, a process blocked by semWait is unblocked

# A Definition of a Counting Semaphore

```
struct semaphore {
    int count;
    queueType queue;
};

void semWait(semaphore* s) {
    s->count--;
    if (s->count < 0) {
        // place this process in s->queue
        // block this process
    }
}

void semSignal(semaphore* s) {
    s->count++;
    if (s->count <= 0) {
        // remove a process P from s->queue
        // place process P on ready list
    }
}
```

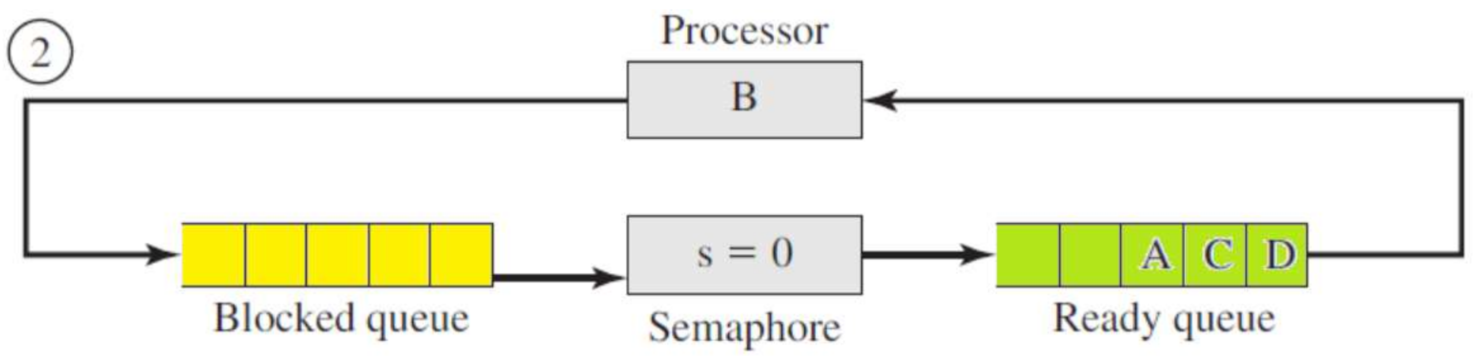
# Examples of Semaphore Mechanism

- Process A, B, C call semWait(s)
- Process D calls semSignal(s)



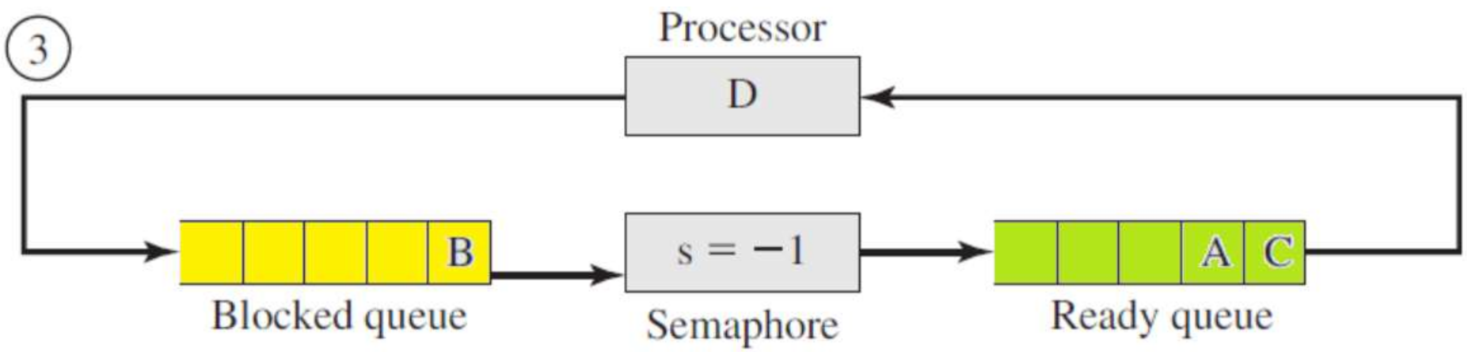

- Initially s is 1
- A will call semWait(s)

# Examples of Semaphore Mechanism



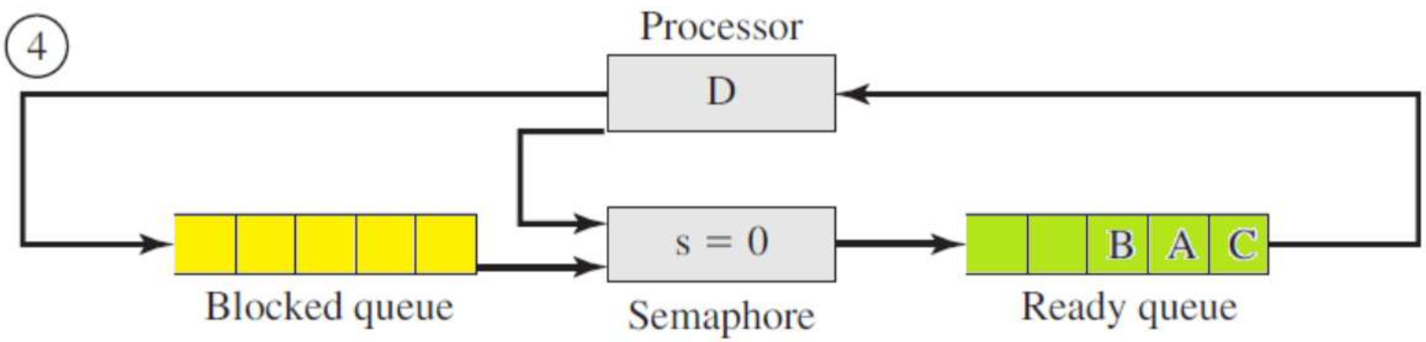

- s is decreased to 0
- A is placed in the Ready queue
- B will call semWait(s)

# Examples of Semaphore Mechanism



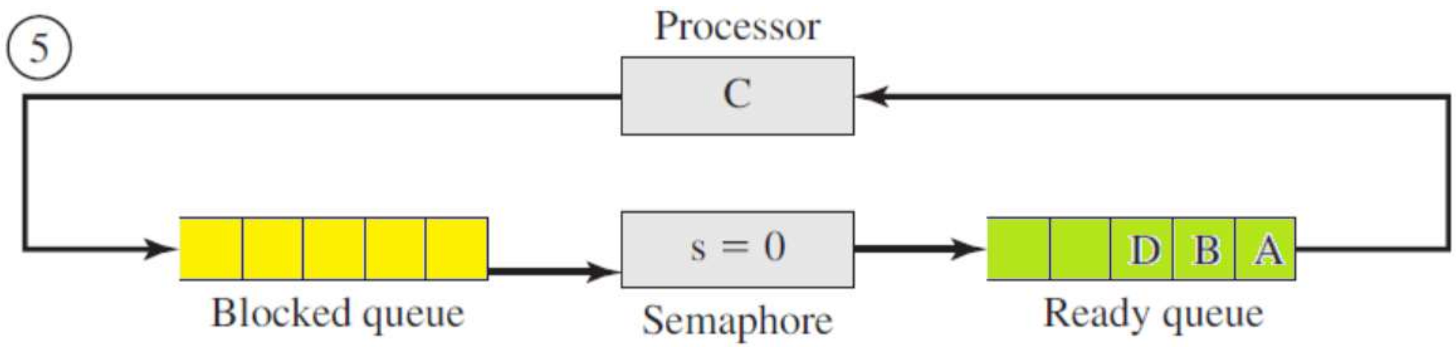

- s is decreased to -1
- B is placed in the semaphore's Blocked queue
- D will call semSignal(s)

# Examples of Semaphore Mechanism



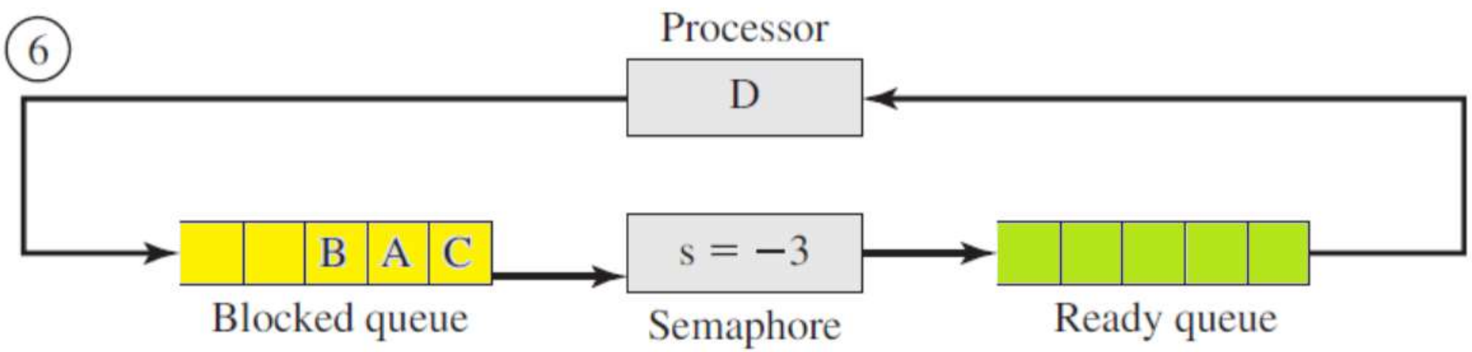

- s is increased to 0
- B is unblocked and is placed in the Ready queue

# Examples of Semaphore Mechanism



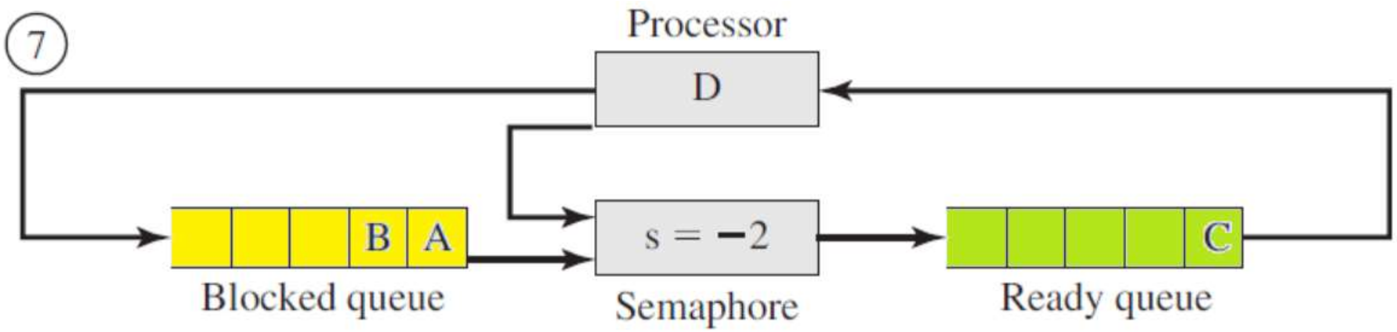

- D is placed in the Ready queue
- C, A, B will call semWait(s)

# Examples of Semaphore Mechanism




- s is decreased to -3
- C, A, B are placed in the semaphore's Blocked queue
- D will call semSignal(s)

# Examples of Semaphore Mechanism




- s is increased to -2
- C is unblocked and is placed in the Ready queue

# Binary Semaphores

- Binary semaphore operations
  - Initialization: a binary semaphore may be initialized to 0 or 1
  - semWaitB
    - If the value is 0, block the process
    - Otherwise, change the value to 0
  - semSignalB
    - If any processes are blocked on this semaphore, one of the blocked processes is unblocked
    - Otherwise, change the value to 1

# A Definition of Binary Semaphore

```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};

void semWaitB(binary_semaphore* s) {
    if(s->value == one)
        s->value = zero;
    else {
        // place this process in s->queue
        // block this process
    }
}

void semSignalB(binary_semaphore* s) {
    if(/*s->queue is empty*/)
        s->value = one;
    else {
        // remove a process P from s->queue
        // place process P on ready list
    }
}
```

# Semaphores

- Some related terms
  - Mutual exclusion lock (mutex):
    - In some literature, mutexes are a synonym for binary semaphores
    - In others, mutexes are like binary semaphores, but with the requirement that the process that locks a mutex must unlock it.
  - Strong semaphore:
    - Processes are blocked and unblocked in FIFO manner
  - Weak semaphore:
    - Any process blocked on the semaphore can be unblocked

# Mutual Exclusion by Semaphores

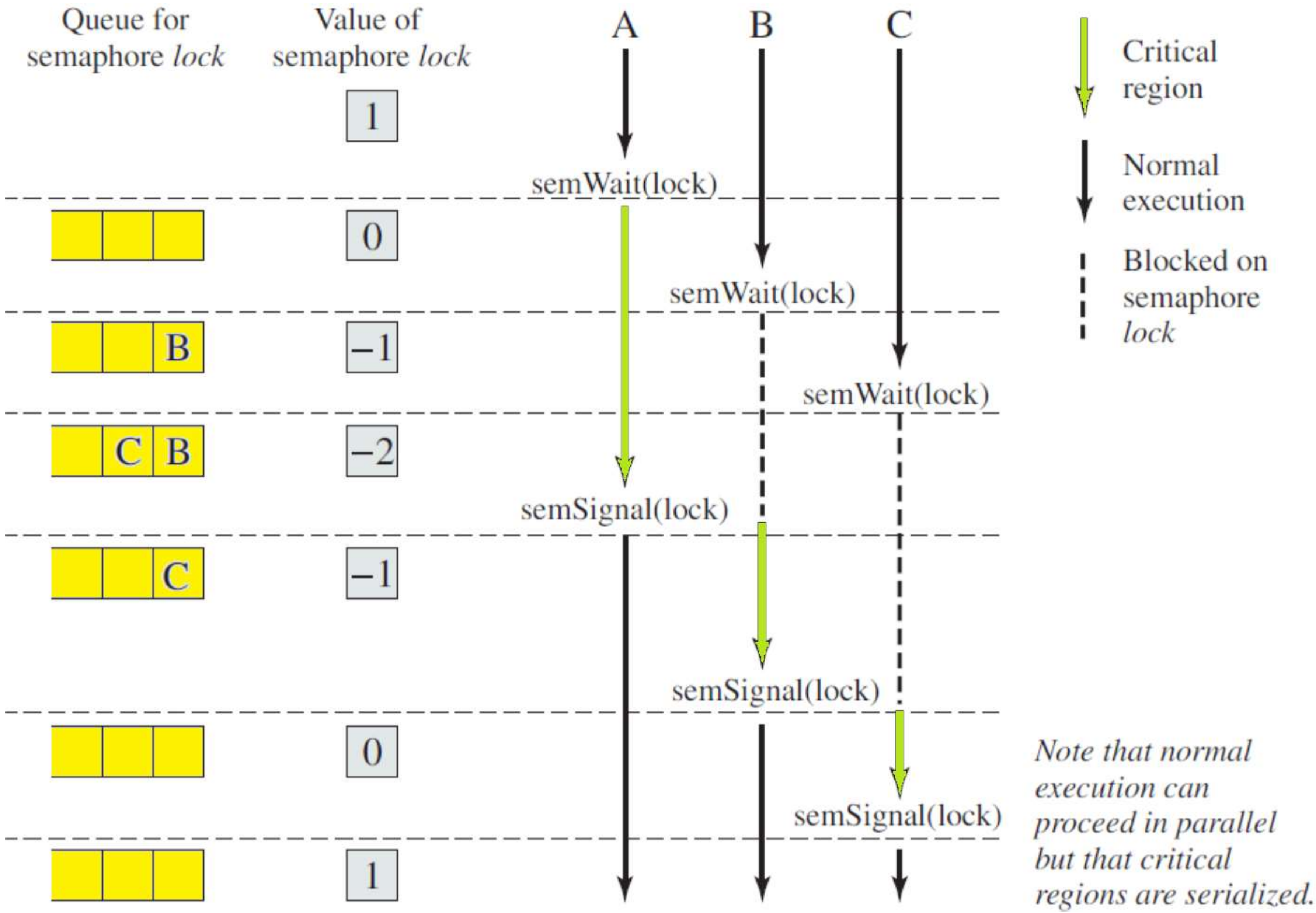
- Initialize a semaphore s to 1
- On entering a critical section call semWait(s)
- On leaving the critical section call semSignal(s)

- Advantages
  - No busy waiting
  - Works with multiple processes on multiple processors

# Mutual Exclusion by Semaphores

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

sem_t mutex;
volatile long count = 0;

void* acc(void *vargp)
{
    long n = *((long*)vargp);
    long i;
    for(i = 0; i < n; i++)
    {
        sem_wait(&mutex);
        count++;
        sem_post(&mutex);
    }
    return NULL;
}
```

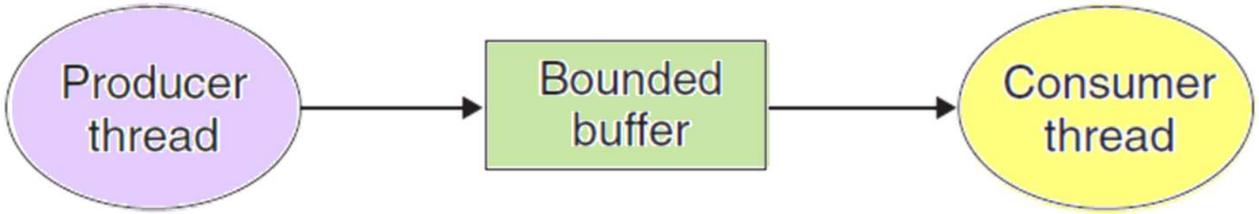
```c
int main()
{
    pthread_t tid1, tid2;
    long n = 100000;

    sem_init(&mutex, 0, 1);

    pthread_create(&tid1, NULL, acc, &n);
    pthread_create(&tid2, NULL, acc, &n);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

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




Accessing shared data protected by a semaphore

# Producer-Consumer Problem

- A producer and a consumer thread share a bounded buffer with n slots
  - The producer creates items and add them to the buffer
  - The consumer removes items from the buffer and consumes (uses) them



# 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 {
    int *buf;
    int capacity, head, tail;

    sem_t mutex; // to access this buffer exclusively
    sem_t slots; // # of empty slots. Block producer if buffer is full
    sem_t items; // # of items. Block consumer if buffer is empty
} 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);
}
void sbuf_deinit(sbuf_t *sp) {
    free(sp->buf);
    sem_destroy(&sp->mutex);
    sem_destroy(&sp->slots);
    sem_destroy(&sp->items);
}
```

SUNY Korea

```c
int sbuf_size(sbuf_t *sp) {
    sem_wait(&sp->mutex); // access lock
    int n = (sp->head + sp->capacity - sp->tail) % sp->capacity;
    sem_post(&sp->mutex);
    return n;
}

void sbuf_insert(sbuf_t *sp, int item) {
    sem_wait(&sp->slots); // block if the buffer is full
    sem_wait(&sp->mutex); // access lock
    sp->head = (sp->head + 1) % sp->capacity;
    sp->buf[sp->head] = item;
    sem_post(&sp->mutex);
    sem_post(&sp->items); // unblock consumer if it’s been suspended
}

int sbuf_remove(sbuf_t *sp) {
    sem_wait(&sp->items); // block if the buffer is empty
    sem_wait(&sp->mutex); // access lock
    sp->tail= (sp->tail + 1) % sp->capacity;
    int item = sp->buf[sp->tail];
    sem_post(&sp->mutex);
    sem_post(&sp->slots); // unblock producer if it’s been 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++)
            sbuf_insert(sp, j),
            s += 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

- A collection of concurrent threads access 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 a permission to update the object

- Second readers-writers problem (favors writers)
  - Once a writer is ready to write, it performs its operation as soon as possible.
  - A reader that arrives before a writer must wait, if the writer is waiting

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

typedef struct {
    sem_t mutex;      // access lock
    sem_t wlock;      // block reader if a writer is already writing
                      // block writer if there is any reader
    int readercount; // # of readers
} rwlock;

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

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

```c
void acquire_reader_lock(rwlock *lock) {
    sem_wait(&lock->mutex);       // access lock
    lock->readercount++;
    if(lock->readercount == 1)    // if I am the first reader,
        sem_wait(&lock->wlock);   // block myself if a writer is writing
                                  // otherwise, make future writers block
    sem_post(&lock->mutex);
}

void release_reader_lock(rwlock *lock) {
    sem_wait(&lock->mutex);       // access lock
    lock->readercount--;
    if(lock->readercount == 0)    // if I am the last reader,
        sem_post(&lock->wlock);   // unblock any suspended writer
    sem_post(&lock->mutex);
}

void acquire_writer_lock(rwlock *lock) {
    sem_wait(&lock->wlock);
}

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

```c
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;
        release_reader_lock(&pobj->lock);

        printf("R_%d: data: %d\n", i, data);
    }
}

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;
        release_writer_lock(&pobj->lock);

        printf("W_%d: data: %d\n", i, data);
    }
}
```

```c
int main()
{
    pthread_t tid;
    object obj;
    obj.data = 0;
    rwlock_init(&obj.lock);

    pthread_create(&tid, 0, reader, &obj);
    pthread_create(&tid, 0, reader, &obj);
    pthread_create(&tid, 0, writer, &obj);

    pthread_exit(NULL);
}
```

# Condition Variables

- Condition variables
  - Allow threads to block when some conditions do not met
  - Blocking thread is waiting for other threads to do some work
- Mutexes
  - Ensure critical section
  - When waiting on a conditional variable, an associated mutex lock will be released
- pthread_cond_wait
  - Condition variables are almost always used with mutexes
  - Blocks the calling thread and release the mutex it holds

# Condition Variables

- pthread_mutex methods

| Thread call | Description |
|---|---|
| Pthread_mutex_init | Create a mutex |
| Pthread_mutex_destroy | Destroy an existing mutex |
| Pthread_mutex_lock | Acquire a lock or block |
| Pthread_mutex_trylock | Acquire a lock or fail |
| Pthread_mutex_unlock | Release a lock |

# Condition Variables

- pthread_cond methods

| Thread call | Description |
|---|---|
| Pthread_cond_init | Create a condition variable |
| Pthread_cond_destroy | Destroy a condition variable |
| Pthread_cond_wait | Block waiting for a signal |
| Pthread_cond_signal | Signal another thread and wake it up |
| Pthread_cond_broadcast | Signal multiple threads and wake all of them |

```c
//Producer Consumer Problem (with condition variables)
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {
    int *buf;
    int capacity, head, tail;
    pthread_mutex_t mutex;  //to access this buffer exclusively
    pthread_cond_t condc;   //condition variable for the consumer
    pthread_cond_t condp;   //condition variable for the producer
} 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;

    pthread_mutexattr_t attr;   //to allow locking from the owning thread
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&sp->mutex, &attr);

    pthread_cond_init(&sp->condc, NULL/*attr*/);
    pthread_cond_init(&sp->condp, NULL/*attr*/);
}
```

```c
void sbuf_deinit(sbuf_t *sp) {
    free(sp->buf);
    pthread_cond_destroy(&sp->condp);
    pthread_cond_destroy(&sp->condc);
    pthread_mutex_destroy(&sp->mutex);
}

int sbuf_size(sbuf_t *sp) {
    pthread_mutex_lock(&sp->mutex);

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

    pthread_mutex_unlock(&sp->mutex);
    return n;
}
```

```c
void sbuf_insert(sbuf_t *sp, int item) {
    pthread_mutex_lock(&sp->mutex);

    while(sbuf_size(sp) == sp->capacity -1) //wait while the buffer is full
        pthread_cond_wait(&sp->condp, &sp->mutex);
    sp->head = (sp->head+1)%sp->capacity;
    sp->buf[sp->head] = item;
    pthread_cond_signal(&sp->condc);        //wake up the consumer

    pthread_mutex_unlock(&sp->mutex);
}

int sbuf_remove(sbuf_t *sp) {
    pthread_mutex_lock(&sp->mutex);

    while(sbuf_size(sp) == 0)              //wait while the buffer is empty
        pthread_cond_wait(&sp->condc, &sp->mutex);
    sp->tail= (sp->tail+1)%sp->capacity;
    int item = sp->buf[sp->tail];
    pthread_cond_signal(&sp->condp);       //wake up the producer

    pthread_mutex_unlock(&sp->mutex);
    return item;
}
```

# Monitors

- Semaphores are difficult to manage
  - semWait and semSignal can be scattered throughout a program

- Monitors
  - Provide a synchronization mechanism
  - Object-Oriented-Programming-like language construct
  - Consist of
    - Local data and condition variables
    - Procedures
    - Initialization sequence

# Monitors

- Characteristics
  - Local variables are accessed only by the monitor's procedures
  - A process enters the monitor by invoking on its procedures
  - Only one process may be executing in the monitor at a time
    - Other processes are blocked until the monitor becomes available
    - Mutual exclusion is provided by this discipline

# Monitors

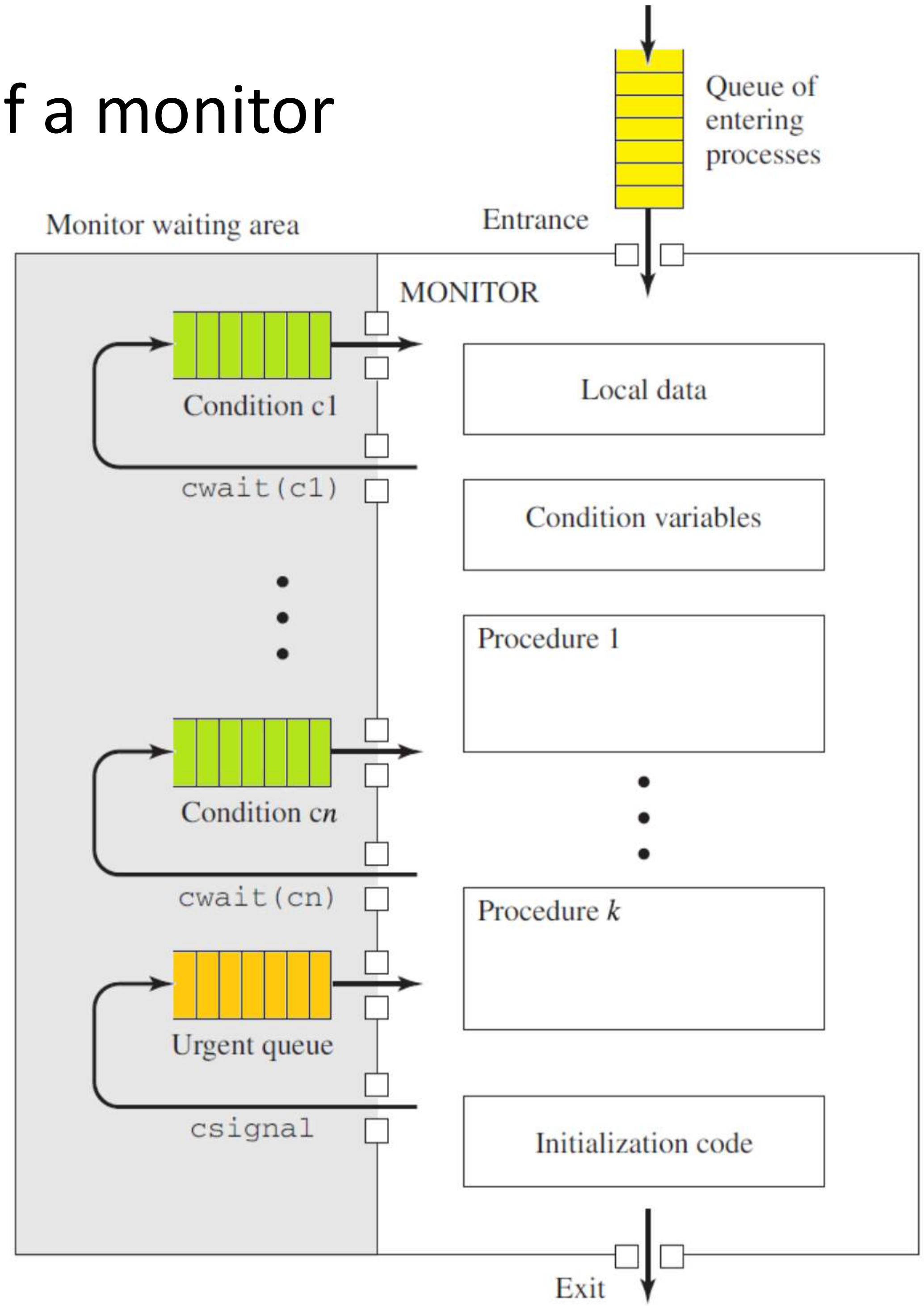
- Condition variables provide synchronization

- cwait(c):
  - Suspend the execution of the process and place it in c's wait queue
  - The monitor is now available for other processes

- csignal(c):
  - Resume the execution of a blocked process from c's wait queue

- cwait and csignal are different from those of semaphores: if c's queue is empty the signal is lost

# Structure of a monitor



# Producer-Consumer by Monitor

```
/* PRODUCER CONSUMER */
void producer() {
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}

void consumer() {
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}
```

```
monitor boundedbuffer;
char buffer[N];                          // buffer with N items
int  nextin, nextout;                    // buffer pointers
int  count;                              // # of items in buffer
cond notfull, notempty;                  // condition variables

void append(char x) {
    if(count == N) cwait(notfull);       // buffer is full, wait on notfull
    buffer[nextin] = x;                  // insert
    nextin = (nextin + 1) % N;
    count++;
    csignal(notempty);                   // resume any waiting consumer
}

void take(char x) {
    if(count == 0) cwait(notempty);      // buffer is empty, wait on notempty
    x = buffer[nextout];                 // remove
    nextout = (nextout + 1) % N;
    count--;
    csignal(notfull);                    // resume any waiting producer
}

// initialization code
{
    nextin = nextout = count = 0;
}
```

# Monitors

- If csignal is not at the end of a procedure
  - The resumed process should run, but there supposed to be only 1 process executing in the monitor
  - The current process can be placed in the entrance queue
  - Or, the current process can be placed in the urgent queue that has a higher priority than the entrance queue

# Monitors with Notify and Broadcast

- cnotify(c)
  - The current process continues to execute
  - A process at the c's condition queue will be resumed at a future time when the monitor is available
- cbroadcast(c)
  - The current process continues to execute
  - All processes at the c's condition queue will be resumed at a future time when the monitor is available

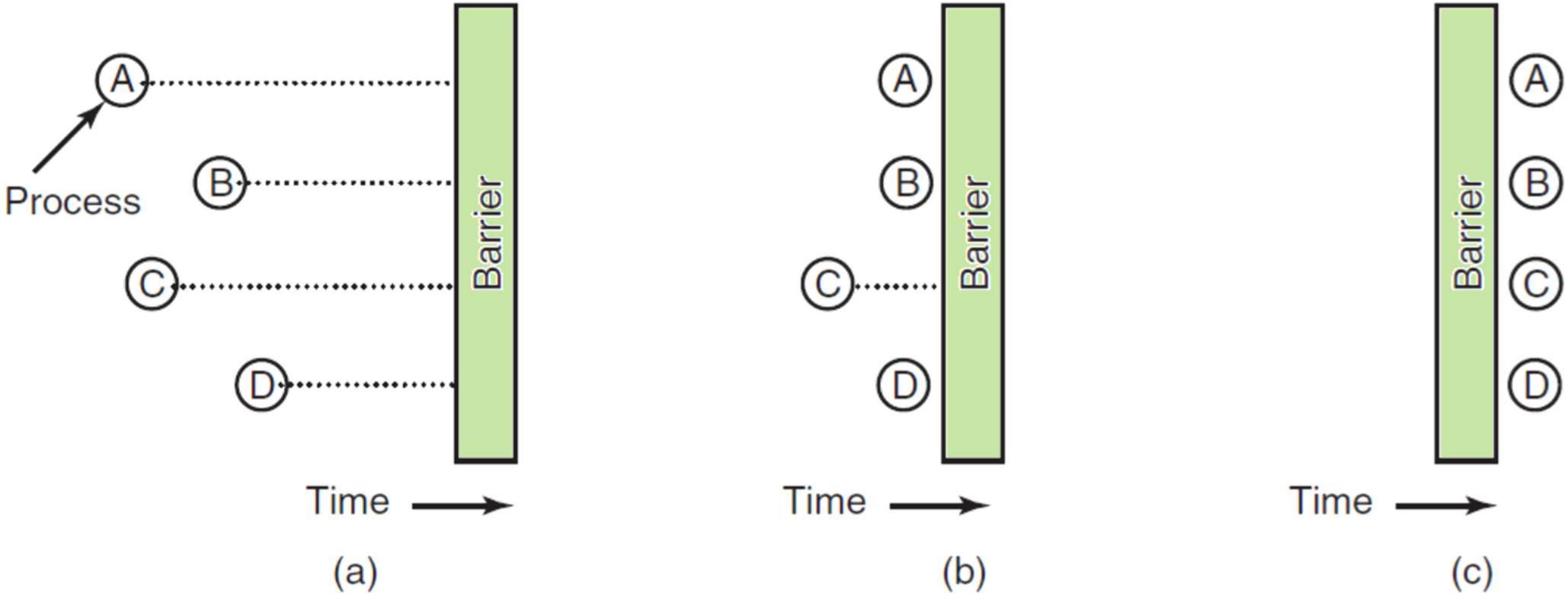
# Monitors with Notify and Broadcast

```
// Producer-consumer for monitor with cnotify

void append(char x) {
    while(count == N) cwait(notfull); // if is replaced by while
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    cnotify(notempty);                 // cnotify instead of csignal
}

void take(char x) {
    while(count == 0) cwait(notempty); // if is replaced by while
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    cnotify(notfull);                  // cnotify instead of csignal
}
```

# Barriers

- Synchronizing a group of processes
  - Some applications are divided into phases
  - No process may proceed to the next phase until all processes finish the current phase
- Put barrier at the end of each phase
  - When a process reaches the barrier, it is blocked until all processes have reached the barrier

# Barriers




a) Processes approaching a barrier
b) All processes but one blocked at the barrier
c) When the last process arrives the barrier, all of them are let through

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

#define MAX_THREAD  10
#define USE_BARRIER 1

pthread_barrier_t barrier;

void* thread(void *vargp) {
    int id = (int)(long)vargp;

    printf("thread %d: enter\n", id);
    for(long i = 0; i < 1000000; i++)
        /*counting 1 million*/;

    printf("thread %d: before barrier\n", id);
#if USE_BARRIER
    pthread_barrier_wait(&barrier);
#endif
    printf("thread %d: after barrier\n", id);

    return NULL;
}
```

```c
int main() {
    pthread_t ids[MAX_THREAD];

    pthread_barrier_init(&barrier, NULL, MAX_THREAD);

    for(int i = 0; i < MAX_THREAD; i++)
        pthread_create(&ids[i], NULL, thread, (void*)(long)i);

    for(int i = 0; i < MAX_THREAD; i++)
        pthread_join(ids[i], NULL);

    pthread_barrier_destroy(&barrier);

    return 0;
}
```