Concurrency: Mutual Exclusion and Synchronization (cont.)

Outline

- Spinlocks and Sleeping locks
- Semaphore
- Producer/consumer problem
- Bounded buffer
- Monitor
- Pthread condition variables

Spinlock

- Construct spinlock with special atomic machine instructions

```
struct{
   int flag;
} lock t;
```

```
void init(lock t *L) {
   L->flag = 0;
}
```

```
void lock(lock t *L) {
    while (compare_and_swap(&lock->flag,0,1)==1){;}
}

void unlock(lock t *L) {
    L->flag = 0;
}
```

- It is called spinlock because of busy-waiting

Spinlock

- Spinlock guarantees mutual exclusion
  - Spinlocks may be used when the critical section is short
  - Spinning may be acceptable when the number of threads is not more than the number of cores in a multicore system
- However, busy-waiting not only wastes CPU cycles, may also cause unbounded waiting on a single CPU system
  - Assume there are two threads t1 and t2 and the priority of t1 is higher
    - t1 is interrupted (e.g., I/O), t2 then starts execution and acquires a lock
    - t1 returns from the interrupt and gets the CPU again
    - When t1 wants to get the lock, the lock has already held by t2
    - However, t2 has a lower priority, it is unable to run and release the lock
    - As a result, t1 keeps busy waiting and never gets the lock
  - This is a starvation situation!

Sleeping Lock

- To get rid of busy waiting (mostly)
  - To lock, suspend the calling thread, change its state and insert it to a waiting queue if the mutex is locked
  - To unlock, remove one thread from the waiting queue and insert it to the ready queue if the waiting queue for the lock is not empty
- Different systems have different solutions
  - E.g., pack/unpack system calls in Solaris and futex system call in linux

futex

- Linux provides the futex system call to optimize the performance of sleeping lock
- futex interface:
  - `void futex_wait(void* addr1, int val)`
    - Calling thread is blocked if `*addr1 == val`
  - `void futex_wake(void* addr1, int n)`
    - Wakes up at most n threads waiting on addr1
    - Typical usage: `n=1 or n=INT_MAX` (broadcast)

Sleeping Lock with futex

- Limit the number of system calls, since changing thread state involves context switch which is costly
  - E.g., There is no contention on the lock, i.e., only a single thread tries to access its critical section
  - should seek for a solution which is optimized for this case
- When there is no contention for the lock with only one thread acquiring and releasing a lock, futex_wait and futex_wake will not be called
  - lock: 1 atomic operation + 0 system call
  - unlock: 1 atomic operation + 0 system call
- There is an example in the textbook

Busy-Waiting vs Sleeping

- Each approach is better under different circumstances
- Spin or block depends on how long before lock is released
  - Lock released quickly -> busy-waiting
  - Lock released slowly -> sleeping
  - Quick and slow are relative to context-switch cost

Two-Phase Waiting

- Theory: Bound worst-case performance
  - ratio of actual/optimal
- When does worst-possible performance occur?
  - Spin for very long time t >> cost of context switch C
  - Ratio: t/C (unbounded)
- Spin-wait for C then block -> Factor of 2 of optimal
  - Two cases:
    - t < C: optimal spin-waits for t; we spin-wait t too
    - t > C: optimal blocks immediately (cost of C);
      - we pay spin C then block (cost of 2C);
      - 2C / C -> 2-competitive algorithm

Synchronization Objectives

- Mutual exclusion (e.g., A and B don’t run at same time)
  - solved with locks

```
lock(l);
//critical section;
…
unlock(l);
```

- Ordering (e.g., B runs after A does something)
  - solved with semaphores and condition variables

Semaphore

- A queue is used to hold processes/threads waiting on the semaphore – eliminating busy-wait
- Semaphore:
  - An integer value used for signalling among processes/threads
- Only three operations may be performed on a semaphore, all of which are atomic:
  - Initialize the integer, (`sem_init`)
  - decrement the value (`sem_wait`)
  - increment the value (`sem_post`)

Semaphore

- Initialize

```
sem_init(sem_t *s, int initval) {
    s->value = initval;
    ...
}
```

  - User cannot read or write value directly after initialization
- Wait: `sem_wait(sem_t*)`
  - Decrement sem value by 1, Waits if value of sem is negative (< 0)
- Post: `sem_post(sem_t*)`
  - Increment sem value by 1, then wake a single waiter if exists
- Value of the semaphore, when negative = the number of waiting threads

Implementation of Semaphores

```
struct semaphore {
    int count;
    queue_type queue;
};
void sem_wait (semaphore s) {
    s.count--;
    if (s.count < 0) {
        //insert calling process to s.queue
        //block the calling process
}
void sem_post (semaphore s) {
    s.count++;
    if (s.count <= 0) {
        //awaken one process from s.queue
        //insert awakened process to ready queue
}
```

Implementation of Semaphores

- Note: the sem_wait and sem_post operations themselves are critical sections and thus must be implemented as atomic primitives
- Use one of the special hardware instructions for mutual exclusion
  - Need busy-waiting, but the critical section is short

Binary Semaphore

- Semaphore can be used as a mutex when `s` is initialized to 1

```
sem_t mtx;
sem_init(&mtx, 1); //initialize s to 1
...
sem_wait(&mtx); //lock
//critical section
...
sem_post(&mtx);//must unlock
```

Enforce Execution Order

To enforce an order: (e.g., S1 must be executed before S2)

```
Sem_init(&s, 0); // initialize s to 0
thrd_0(){                   thrd_1(){
    ...                         ...
    S1;                         sem_wait (s);
    sem_post (s);               S2;
    ...                         ...
}                           }
```

Producer/Consumer Problem

- General Situation:
  - One or more producers are generating data and placing these in a buffer of size N
  - One or more consumers are taking items out of the buffer one at time
- The Problem:
  - Ensure that consumers can't remove data from an empty entry and producers can't add data into a nonempty entry in the buffer

Producer/Consumer Problem

- First consider a very simple case:
  - Single producer thread, single consumer thread
  - Single shared buffer between producer and consumer
- Use 2 semaphores
  - `emptyBuffer`: Initialize to 1
  - `fullBuffer`: Initialize to 0

Producer thread:

```
while (1) {
   sem_wait(&emptyBuffer);
   fill(&buffer);
   sem_post(&fullBuffer);
}
```

Consumer thread:

```
while (1) {
   sem_wait(&fullBuffer);
   take(&buffer);
   sem_post(&emptyBuffer);
}
```

Producer/Consumer Problem

- Now consider a bit more complicated (still simple) case:
  - Still, single producer thread, single consumer thread
  - But, shared buffer with N elements between producer and consumer
- Use 2 semaphores
  - `emptyBuffer`: Initialize to N
  - `fullBuffer`: Initialize to 0

Producer thread:

```
i = 0;
while (1) {
   sem_wait(&emptyBuffer);
   fill(&buffer[i]);
   i = (i+1)%N;
   sem_post(&fullBuffer);
}
```

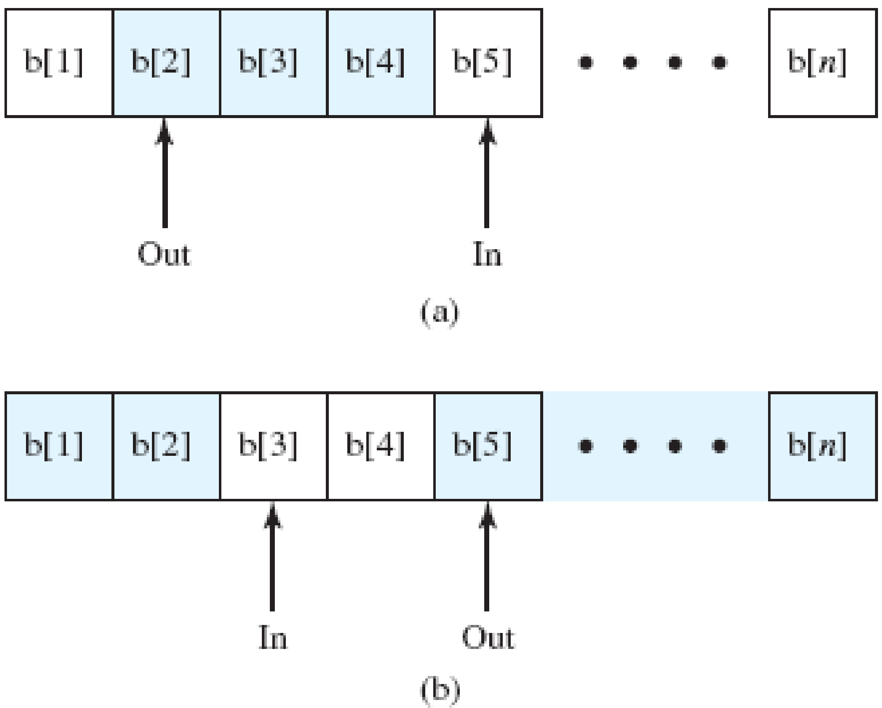
Consumer thread:

```
j = 0;
while (1) {
   sem_wait(&fullBuffer);
   take(&buffer[j]);
   j = (j+1)%N;
   sem_post(&emptyBuffer);
}
```

Producer/Consumer Problem

- Now consider more general case:
  - Multiple producer threads, multiple consumer threads
  - Shared buffer with N elements between producer and consumer
- Requirements
  - Each consumer must grab unique filled element
  - Each producer must grab unique empty element

Producer/Consumer Problem



Producer/Consumer Problem

- Assume that we have two functions
  - findempty(&buffer): find the first empty entry in the buffer
  - findfull(&buffer): find the first nonempty entry in the buffer

- Producer thread:

```
while (1) {
   sem_wait(&emptyBuffer);
   my_i = findempty(&buffer);
   fill(&buffer[my_i]);
   sem_post(&fullBuffer);
}
```

Consumer thread:

```
while (1) {
   sem_wait(&fullBuffer);
   my_j = findfull(&buffer);
   take(&buffer[my_j]);
   sem_post(&emptyBuffer);
}
```

- Problem: buffer is shared – potential race condition

Producer/Consumer Problem

- To prevent race condition, need to use a mutex

- Producer thread:

```
 while (1) {
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    my_i = findempty(&buffer);
    sem_post(&mutex);
    fill(&buffer[my_i]);
    sem_post(&fullBuffer);
}
```

Consumer thread:

```
while (1) {
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    my_j = findfull(&buffer);
    sem_post(&mutex);
    take(&buffer[my_j]);
    sem_post(&emptyBuffer);
}
```

- This solution has a problem! – why?

Producer/Consumer Problem

- Producer thread:

```
while (1) {
   sem_wait(&emptyBuffer);
   sem_wait(&mutex);
   my_i = findempty(&buffer);
   fill(&buffer[my_i]);
   sem_post(&mutex);
   sem_post(&fullBuffer);
}
```

Consumer thread:

```
while (1) {
   sem_wait(&fullBuffer);
   sem_wait(&mutex);
   my_j = findfull(&buffer);
   take(&buffer[my_j]);
   sem_post(&mutex);
   sem_post(&emptyBuffer);
}
```

- This one is correct!

Issues with Semaphores

- Semaphores provide a powerful synchronization tool
  - Can be used for both mutual exclusion and ordering
- However,
  - Sem_post() and sem_wait() are scattered among several processes/threads in complicated programs
    - Therefore, it is difficult to understand their effects
  - Usage must be correct in all the processes/threads
  - One bad process/thread (or one programming error) can kill the whole system

Issues with Semaphores

- 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

P_0	P_1
sem_wait (&S);	sem_wait (&Q);
sem_wait (&Q);	sem_wait (&S);
…	…
sem_post (&S);	sem_post (&Q);
sem_post (&Q);	sem_post (&S);

Monitors

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control in many applications
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Pascal-Plus,Modula-2, Modula-3, and Java
- Software module consisting of one or more procedures, an initialization sequence, and local data

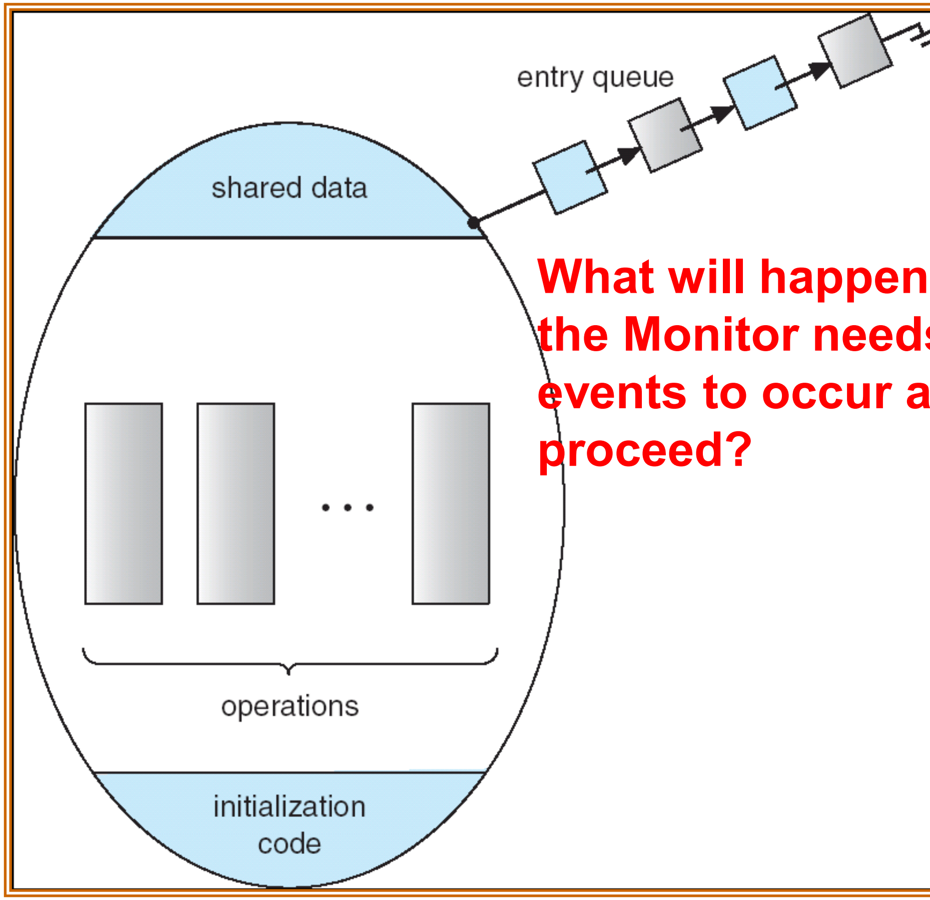
Monitor Characteristics

Local data variables are accessible only by the monitor's procedures and not by any external procedure

Process enters monitor by invoking one of its procedures

Only one process may be executing in the monitor at a time

Schematic View of a Monitor

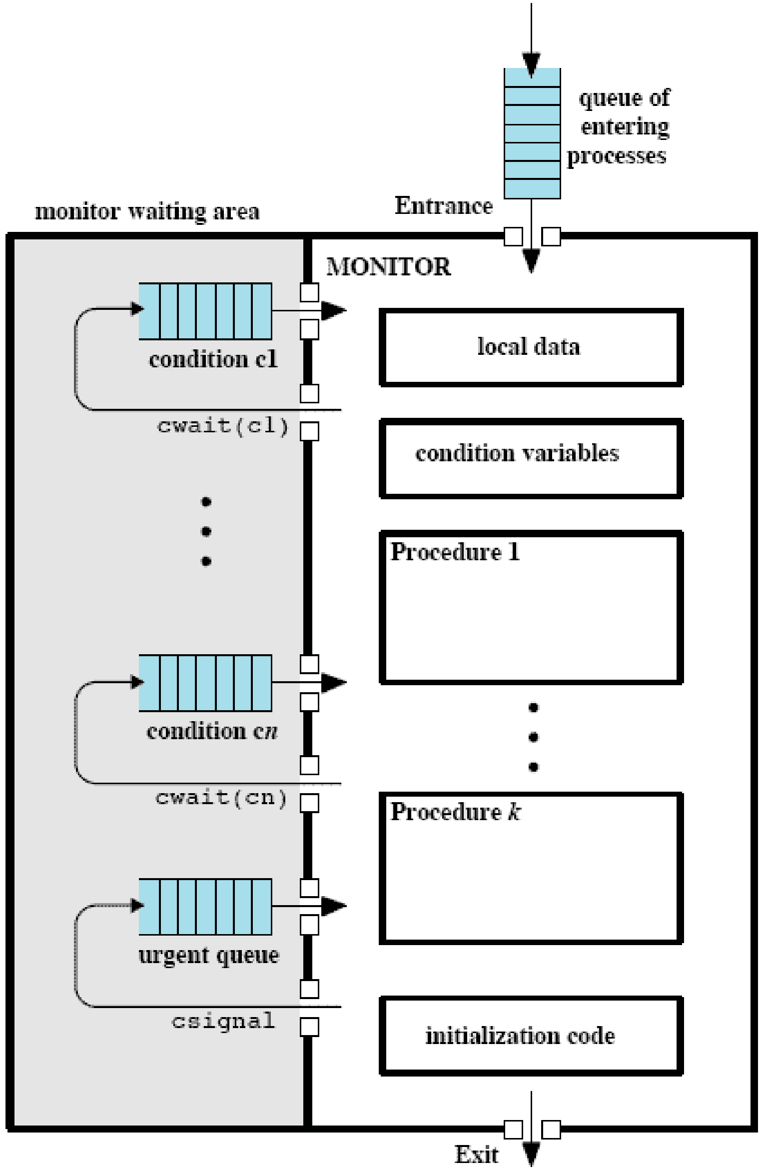


What will happen if one process in the Monitor needs to wait for some events to occur and is unable to proceed?

Synchronization

- Synchronisation achieved by using **condition variables** that are contained within a monitor and only accessible within the monitor
- Condition variables are operated on by two functions:
  - `cwait(c)`: Suspend execution of the calling process on condition *c*
  - `csignal(c)` Resume execution of one process blocked after a `cwait(c)` on the same condition

Structure of a Monitor



Pthread Condition Variables

- Pthread specifies condition variables, but not monitors
  - Pthread was originally designed for C programming language
  - C is not an object-oriented programming language
- A condition variables must be used in conjunction with a mutex lock

Pthread Condition Variables

- Three rountines to wait/signal on condition variables:
  - `pthread_cond_wait(*condition, *mutex)`
  - `pthread_cond_signal(*condition)`
  - `pthread_cond_broadcast(*condition)`

- `pthread_cond_wait()` blocks the calling thread until the specified condition is signalled
- This routine should be called while mutex is locked, and it will automatically release the mutex while it waits
- After signal is received and one blocked thread is awakened, mutex will be automatically locked for use by the thread
- The programmer is then responsible for unlocking mutex when the thread is finished with it

Pthread Condition Variables

- Three rountines to wait/signal on condition variables:
  - `pthread_cond_wait(*condition, *mutex)`
  - `pthread_cond_signal(*condition)`
  - `pthread_cond_broadcast(*condition)`

- `pthread_cond_signal()` is used to signal (or wake up) another thread which is waiting on the condition variable
- It should be called after mutex is locked and must unlock mutex in order for `pthread_cond_wait()` routine to complete.

- `pthread_cond_broadcast()` routine unlocks all of the threads blocked on the condition variable

Pthread Condition Variables

- Proper locking and unlocking of the associated mutex variable is essential when using these routines
- Failing to lock the mutex before calling `pthread_cond_wait()` may cause it NOT to block
- `pthread_cond_signal()` and `pthread_cond_wait()` must be called between `pthread_mutex_lock(&m)` and `pthread_mutex_unlock(&m)`
  - Consider monitors!

Producer/Consumer Problem

- Buffer size: `N`;
- A count: `numfull`; how many elements currently in the buffer
- Need two condition variables:
  - `empty`; if the buffer is full, producer wait
  - `full`; if the buffer is empty, consumer wait
- Function `findempty()`; find the first empty entry and also increment `numfull`
- Function `findfull()`; find the first nonempty entry and also decrement `numfull`

Producer/Consumer Problem

```
void *producer(void *arg){
  while(1)){
    pthread_mutex_lock(&m);
    while (numfull == N) //use while, but not if!
      pthread_cond_wait(&empty , &m);
    my_i = findempty(&buffer);
    fill(&buffer[my_i]);
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&m);
  }
}
```

Producer/Consumer Problem

```
void *consumer(void *arg){
  while(1){
    pthread_mutex_lock(&m);
    while(numfull == 0) //again use while, but not if!
      pthread_cond_wait(full, &m);
    my_j = findfull(&buffer);
    take(&buffer[my_j]);
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&m);
  }
}
```

Summary

- Locks for mutual exclusion
  - Spinlocks vs sleeping locks
  - Two-phase waiting
- Semaphores
  - Used for signalling among processes/threads and can be readily used to enforce a mutual exclusion discipline
- Monitors
  - A programming-language construct that provides equivalent functionality to that of semaphores and sometimes is easier to control
- Pthread condition variables