Operating Systems

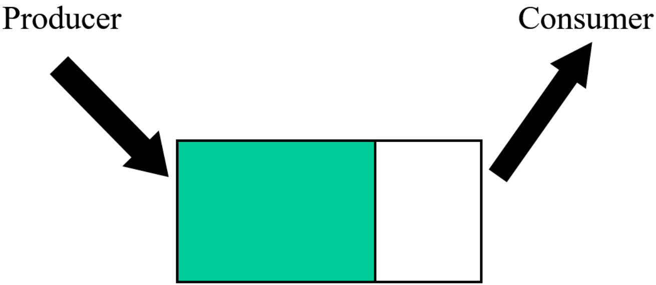
0107451
Chapter 2 Processes and Threads

Dr. Naeem Odat

Tafial Technical University
Department of Computer and Communications Engineering

2.3 The Producer Consumer Problem

Producer Consumer



Issues

- Mutual Exclusion.
- Buffer full.
- Buffer empty.

2.3 The Producer Consumer Problem

Producer Consumer

```
Producer:
    while(True){
        item=produce();
        insert(item, buffer);
        count++;
    }
```

```
Consumer:
    while(True){
        item=remove(buffer);
        count--;
        consume(item);
}
```

Problems?

- How does any of them know about how much in buffer?
- What about critical regions?

2.3 The Producer Consumer Problem

Mutual Exclusion

```
Producer:
    while(True)
        item=produce();
        lock(mutex);
        insert(item, buffer);
        count++;
        unlock(mutex);
```

```
Consumer:
    while(True)
        lock(mutex);
        item=remove(buffer);
        count--;
        unlock(mutex);
        consume(item);
```

Problems?

How does any of them know about how much in buffer?

2.3 The Producer Consumer Problem

Producer Consumer

- Cannot be solved by mutexes only.
- Need a way to block till some condition is satisfied:
  - Sleep and Wakeup
  - Semaphore (not part of pthread package)
  - Condition variables (preferred with pthread)

2.3 The Producer Consumer Problem

- **N**: Size of the buffer
- Shared variables:
  - **count**: number of item in the buffer
  - **buffer**

Sleep and Wakeup

```
Producer:
    while(True)
        item=produce();
        if(count==N)
            sleep();
        insert(item, buffer);
        count++;
        if(count==1)
            wakeup(consumer);
```

```
Consumer:
    while(True)
        if(count==0)
            sleep();
        item=remove(buffer);
        count--;
        if(count==N-1)
            wakeup(producer);
        consume(item);
```

2.3 The Producer Consumer Problem

- **N**: Size of the buffer
- Shared variables:
  - **count**: number of item in the buffer
  - **buffer**

Sleep and Wakeup

```
Producer:
    while(True)
        item=produce();
        if(count==N)
            sleep();
        lock(mutexs);
        insert(item, buffer);
        count++;
        unlock(mutexs);
        if(count==1)
            wakeup(consumer);
```

```
Consumer:
    while(True)
        if(count==0)
            sleep();
        lock(mutexs);
        item=remove(buffer);
        count--;
        unlock(mutexs);
        if(count==N-1)
            wakeup(producer);
        consume(item);
```

2.3 The Producer Consumer Problem

Semaphore Interface

```
S: Integer value

Down(S):
    when(S>0)
        S=S-1;

Up(S):
    S=S+1;
```

- Atomic actions
- Down might block
- Up never blocks

Semaphore Implementation

Down(S):

- If (S==0) then:
  - Suspend thread, put it into a waiting queue
  - Schedule another thread to run
- Else:
  - Decrement S
  - Return

Up(S):

- Increment S
- If any is in waiting queue then:
  - Release one of them (make it runnable)

2.3 The Producer Consumer Problem

- **N**: Size of the buffer
- Shared variables:
  - **count**: number of item in the buffer
  - **buffer**
- **Empty:** semaphore initialized to N (number of empty slots)
- **Full:** semaphore initialized to zero (number of filled slots)

Semaphore

```
Producer:
    while(True)
        item=produce();
        down(Empty);
        lock(mutexs);
        insert(item, buffer);
        count++;
        unlock(mutexs);
        up(Full);
```

```
Consumer:
    while(True)
        down(Full);
        lock(mutexs);
        item=remove(buffer);
        count--;
        unlock(mutexs);
        up(Empty);
        consume(item);
```

2.3 The Producer Consumer Problem

Blocking mutex is a special case of Semaphore

- Initialize semaphore S = 1
- lock_mutex = Down(S)
- unlock_mutex=Up(S)
- Difference:
  - With pthread_mutexes, only the thread which currently holds the lock can unlock it.
  - No such restriction in semaphores.
- It is called a binary semaphore

Example (1) on semaphore usage

Computer game with multiple players, where no more than 2 players in a room.

Solution

- Initialize semaphore S=2
- Players execute:
  - Down(S) before entering.
  - Up(S) while leaving.

Example (2) on semaphore usage

Web Server can handle only 10 threads at a time:

- Multiple points where threads are being created
- How to ensure no more than 10 active threads?

Solution

Semaphore with initial value S = 10

- Down(S) before thread creation
- Up(S) once thread finishes

POSIX Semaphore

```
#include<semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
pshared=0: semaphore is shared between threads in a process.
pshared!=0: semaphore is shared between processes.
value: initial value of the semaphore

int sem_wait(sem_t * sem);//down(s)
int sem_trywait(sem_t * sem);//try to down(s)
int sem_post(sem_t * sem);//up(s)
int sem_getvalue(sem_t * sem, int * sval);
int sem_destroy(sem_t * sem);
```

2.3 The Producer Consumer Problem

Questions

1. What if we changed the order of **lock()** and **Down()** in producer consumer solution?
2. What if we changed the order of **unlock()** and **Up()**?

2.3 The Producer Consumer Problem

Implementing wait() system call with semaphore

- Parent process does a wait() system call on child (wait till child finishes before exiting).
- What if parent executed wait() after child exited? (wait should return immediately).

Implementing wait() system call with semaphore - solution

- Semaphore zombie: initialize to 0
- Parent: down(zombie) inside wait()
- Child: up(zombie) upon exiting.

Condition variables

Allows a thread to wait till a condition is satisfied

Testing if the condition is satisfied must be done within a mutex

With every condition variable, a mutex is associated

2.3 The Producer Consumer Problem

Condition variables

```
pthread_cond_t condition_variable;
pthread_mutex_t mutex;
```

Waiting Thread:

```
pthread_mutex_lock(&mutex);

while(condition not satisfied){
    pthread_cond_wait(
    &condition_variable,
    &mutex);
}
pthread_mutex_unlock(&mutex);
```

Signaling Thread:

```
pthread_mutex_lock(&mutex);
/* Change variable values */
if(condition is satisfied){
    pthread_cond_signal(
    &condition_variable);
}
pthread_mutex_unlock(&mutex);
/* Alternative to cond_signal is
  pthread_cond_broadcast(
  &condition_variable);
*/
```

2.3 The Producer Consumer Problem

Condition variables and mutex

A mutex is passed into wait: pthread_cond_wait(cond_var, mutex)

Mutex is released before the thread sleeps

Mutex is locked again before **pthread_cond_wait()** returns

Safe to use **pthread_cond_wait()** in a while loop and check condition again before proceeding

Usage Example

Write a program using two threads:

- Thread 1 prints "hello"
- Thread 2 prints "world"
- Thread 2 should wait till thread 1 finishes before printing.

Use a condition variable

2.3 The Producer Consumer Problem

Solution

```
int thread1_done = 0;
pthread_cond_t cv;
pthread_mutex_t mutex;
```

```
Thread 1:
    printf("Hello ");

    pthread_mutex_lock(&mutex);
    thread1_done=1;
    pthred_cond_signal(&cv);
    pthread_mutex_unlock(&mutex);
```

```
Thread 2:
    pthread_mutex_lock(&mutex);

    while(thread1_done==0){
        pthread_cond_wait(&cv, &mutex);
    }
    printf("world\n");
    pthread_mutex_unlock(&mutex);
```