Thread Synchronization

CS 360 Internet Programming

Daniel Zappala

Brigham Young University
Computer Science Department

A Simple Example

```
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

- shared method among multiple threads
- unpredictable execution sequence
  - Thread1 executes up to conclusion of input function, then interrupted: Thread1 reads 'x'
  - Thread2 executes completely through the procedure: Thread2 reads and prints 'y'
  - Thread1 starts again: Thread1 input variable has 'y'!

Concurrency Problems

- *demonstration*

Concurrency Problems

- *demonstration*
- problems
  - can't predict the speed with which threads will execute and therefore when a resource will be accessed
  - if synchronization is not used, errors will be rare but they will occur
  - errors are hard to duplicate and debug since they are nondeterministic

Mutual Exclusion

- need to protect shared resources (e.g. global variable, shared data structures) among multiple processes or threads
  - atomic access to method
  - when Thread2 tries to enter method, block it until Thread1 is finished
- may involve processes or threads interleaved in time on a single processor or running in parallel on a multiprocessor machine
- result of process or thread must be independent of the speed of execution of other concurrent processes

Mutual Exclusion

- critical section: shared portion of code that must be executed by one thread at a time
  - thread must mark the critical section because OS doesn't know where it is
- starvation: one or more threads are prevented from ever executing critical section
- deadlock: situation in which no thread can make progress because they are all waiting for a critical section
- must ensure data coherence, e.g. atomic access to a database

Mutual Exclusion Requirements

- only one thread has access to critical section at a time
- halting in non-critical section must not interfere with other threads
- no indefinite wait for critical section, i.e. no starvation or deadlock
- if no thread in critical section, then no wait to enter
- no assumptions about process speeds or number of processors
- thread may only spend finite time within critical section

Solutions

- software
  - assume no support from OS, hardware, or language
  - historic algorithms: Dekker, Peterson, Lamport
  - difficult to get right, to generalize
- hardware
  - disable interrupts: single processor machines
    - no other process can run until they are re-enabled
    - limits flexibility of OS to schedule threads, doesn't work for multiprocessors
  - atomic machine instructions: test-and-set
- operating system support

Test-and-Set Example

```
boolean test-and-set(int i) {
  if (i == 0) {
    i = 1;
    return true;
  } else {
    return false;
  }
}

int bolt = 0;
void method(int i) {
  while (true) {
    while (!test-and-set(bolt))
      /* do nothing */
    /* critical section */
    bolt = 0;
    /* remainder */
  }
}
```

Test-and-Set Pros and Cons

- advantages
  - applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - it is simple and therefore easy to verify
  - it can be used to support multiple critical sections: each section gets its own variable
- disadvantages
- busy-waiting consumes processor time
- starvation is possible when more than one process waits
- deadlock
  - low priority process has the critical region
  - higher priority process needs it
  - higher priority process obtains the processor and waits for the critical region

Operating System Support

- mutex and condition variable
  - mutex: lock that allows only one thread into a critical section
  - condition variable: signal conditions between threads
- semaphore
  - when one thread is in the critical section, others may wait by sleeping
  - when thread is done with critical section, it wakes one other thread with a signal
- monitor
  - programming language construct that makes it easier to declare and use a critical section
  - construct a class with methods, only one thread may access a method of the class at a time
- message passing
  - synchronization by explicitly exchanging messages
  - define a mailbox and enter critical section when a message is waiting

Mutex

- lock that allows only one thread into a critical section

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- must initialize the mutex first
- `pthread_mutex_lock()` will block if mutex is already locked
- `pthread_mutex_trylock()` will return EBUSY if mutex is locked
- *demonstration*

Busy Waiting

```
while running {
  c = NULL;
  pthread_mutex_lock(&mutex);
  if queue.not_empty() {
    c = queue.dequeue();
  }
  pthread_mutex_unlock(&mutex);
  if c {
    /* handle connection */
  }
}
```

- must busy wait until a connection is available
- wastes CPU time on a server that does not handle many connections

Condition Variables

```c
#include <pthread.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t);
```

- must initialize the condition variable first
- `pthread_cond_wait()` will block until the condition is signaled; the thread now owns the mutex as well
- need a corresponding `pthread_cond_signal()` to wake up

Using Condition Variables

```
while running {
  c = NULL;
  pthread_mutex_lock(&mutex);
  while queue.empty() {
    pthread_cond_wait(&cond,&mutex);
  }
  c = queue.dequeue();
  pthread_mutex_unlock(&mutex);
  /* handle connection */
}
```

- process inserting into queue should signal condition when queue goes from empty to having at least one item
- **must re-check queue status when conditional wait returns**
- no guarantee that queue will be empty when you return

Timed Wait and Broadcast Signals

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

- `pthread_cond_timedwait()` needs an absolute time; use `clock_gettime()` and add the length of time you want to wait
- `pthread_cond_broadcast()` wakes up all threads waiting for a signal

Semaphores

- semaphore is a shared integer variable
  - initialized $>= 0$
- `wait(s)`: wait for a signal on semaphore *s*
  - decrements semaphore, blocks if < 0
  - process suspends until signal is sent
  - OS uses a queue to hold waiting processes
- `signal(s)`: transmit a signal to semaphore *s*
  - increments semaphore
  - if $<= 0$ then unblock someone
- `wait()` and `signal()` are atomic operations and cannot be interrupted

Types of Sempahores

- binary semaphore
  - only one process at a time may be in the critical section
- counting semaphore
  - a fixed number of processes > 0 may be in the critical section
- OS determines whether processes are released from queue in FIFO order or otherwise; usually FIFO in order to prevent starvation

Using Semaphores

```
semaphore s = 1;

void thread(int i) {
    while (true) {
        wait(s);
        /* critical section */
        signal(s);
       /* remainder */
    }
}
```

- semaphore protects critical section
- can set s to > 1 to let more than one process in the critical section
  - $s >= 0$: number that can enter
  - $s < 0$: number that are waiting

POSIX Semphores

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
int sem_post(sem_t * sem);
```

- `sem_init()`: sets initial value of semaphore; `pshared = 0` indicates semaphore is local to the process
- `sem_wait()`: suspends process until semaphore is > 0, then decrements semaphore
- `sem_trywait()`: returns EAGAIN if semaphore count is $= 0$
- `sem_post()`: increments semaphore, may cause another thread to wake from `sem_wait()`
- *demonstration*

Producer Consumer Problem

- one or more producers are generating data and placing them in a buffer
- one or more consumers are taking items out of the buffer
- only one producer or consumer may access the buffer at any time

Producer Consumer with Infinite Buffer

producer:

```
while (True) {
    /* produce item v */
    b[in] = v;
    in++;
}
```

consumer:

```
while (True) {
    while (in <= out)
        /* do nothing */;
    w = b[out];
    out++;
    /* consume item w */
}
```

Producer Consumer using Binary Sempahores

```
int n;
sem_t s, delay;
sem_init(&s,0,1);
sem_init(&delay,0,0);
```

producer:

```
while (True) {
      produce();
      sem_wait(&s);
      append();
      n++;
      if (n==1) sem_post(&delay);
      sem_post(&s);
}
```

consumer:

```
int m;
sem_wait(&delay);
while (True) {
      sem_wait(&s);
      take();
      n--;
      m = n;
      sem_post(&s);
      consume();
      if (m==0) sem_wait(&delay)
}
```

Looking at the Code ...

1. *What is the purpose of semaphore s?*

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore delay?*

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore delay?*
3. *Why is semaphore s initialized to 1 but semaphore delay is initialized to 0?*

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore delay?*
3. *Why is semaphore s initialized to 1 but semaphore delay is initialized to 0?*
4. *Why does the consumer need to wait on delay first?*

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore delay?*
3. *Why is semaphore s initialized to 1 but semaphore delay is initialized to 0?*
4. *Why does the consumer need to wait on delay first?*
5. *Why does the producer signal wait only when n == 1?*

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore delay?*
3. *Why is semaphore s initialized to 1 but semaphore delay is initialized to 0?*
4. *Why does the consumer need to wait on delay first?*
5. *Why does the producer signal wait only when n == 1?*
6. *Why does the consumer need a local variable m?*

Important Insights

- two purposes for semaphores
  - mutual exclusion: semaphore *s* controls access to critical section
  - signalling: semaphore *delay* coordinates when the buffer is empty: consumer waits if buffer is empty, producer signals when buffer becomes non-empty
- avoid race conditions
  - *m* keeps a local copy of the data protected by the semaphore so that it can be accessed later
  - reduces amount of processing inside the critical section

Producer Consumer using Counting Sempahores

```
sem_t s, n;
sem_init(&s,0,1);
sem_init(&n,0,0);
```

producer:

```
while (True) {
    produce();
    sem_wait(&s);
    append();
    sem_post(&s);
    sem_post(&n);
}
```

consumer:

```
while (True) {
    sem_wait(&n);
    sem_wait(&s);
    take();
    sem_post(&s);
    consume();
}
```

Looking at the Code ...

1. *What is the purpose of semaphore s?*

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore n?*

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore n?*
3. *Why is semaphore s initialized to 1 but semaphore n is initialized to 0?*

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore n?*
3. *Why is semaphore s initialized to 1 but semaphore n is initialized to 0?*
4. *Why can the producer signal n every time an item is added to the buffer ?*

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore n?*
3. *Why is semaphore s initialized to 1 but semaphore n is initialized to 0?*
4. *Why can the producer signal n every time an item is added to the buffer ?*
5. *Can the producer swap the signals for n and s?*

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore n?*
3. *Why is semaphore s initialized to 1 but semaphore n is initialized to 0?*
4. *Why can the producer signal n every time an item is added to the buffer ?*
5. *Can the producer swap the signals for n and s?*
6. *Can the consumer swap the waits for n and s?*

Important Insights

- more elegant solution, can't do this easily with mutexes
- n: semaphore value is number of items in buffer
  - if $n == 0$, consumer must wait
  - can swap `sem_post(&n);` and `sem_post(&s);` in producer and be OK
  - can't swap `sem_wait(&n);` and `sem_wait(&s);` in consumer: otherwise consumer enters and then waits and deadlocks the producer!
- ordering of semaphore operations is important
- would like programming language support to help with organizing mutual exclusion code: monitors

Producer Consumer Circular Buffer

```
sem_t s, n, e;
sem_init(&s,0,1);
sem_init(&n,0,0);
sem_init(&e,0,BUFFER_SIZE);
```

producer:

```
while (True) {
    produce();
    sem_wait(&e);
    sem_wait(&s);
    append();
    sem_post(&s);
    sem_post(&n);
}
```

consumer:

```
while (True) {
    sem_wait(&n);
    sem_wait(&s);
    take();
    sem_post(&s);
    sem_post(&e);
    consume();
}
```

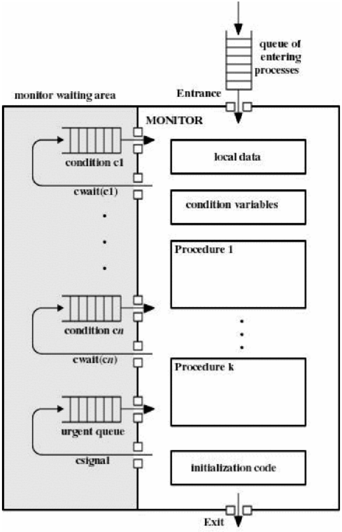
Looking at the Code ...

1. *What is the difference between semaphore e and semaphore n?*

Monitor

- difficult to get semaphores right
  - match wait and signal
  - put in right order
  - scattered throughout code
- monitor: programming language construct
  - equivalent functionality
  - easier to control
  - mutual exclusion constraints can be checked by the compiler
  - used in versions of Pascal, Modula, Mesa
  - Java also has a Monitor object but compliance cannot be checked at compile time

Hoare Monitor



- monitor can only be entered through procedures
- data can only be accessed by procedures
- only one process or thread in monitor at any time
- may suspend and wait on a condition variable
- like object-oriented programming with mutual exclusion added in

Hoare Synchronization

- `cwait(c)`: suspend on condition c
- `csignal(c)`: wake up one thread waiting for condition c
  - do nothing if no threads waiting (signal is lost)
  - different from semaphore (number of signals represented in semaphore value)

Producer Consumer with a Hoare Monitor

```
char buffer[BUFSIZE];
int in = 0, out = 0, count = 0;
condition notfull, notempty;
```

append:

```
if count == N
  cwait(notfull);
buffer[in] = x;
in = (in + 1) % BUFSIZE;
count++;
csignal(notempty);
```

take:

```
if count == 0
    cwait(notempty);
x = buffer[out];
out = (out + 1) % BUFSIZE;
count--;
csignal(notfull);
```

Producer Consumer with a Hoare Monitor

producer:

```
char x;
while (True) {
  x = produce();
  append(x);
}
```

consume:

```
char x;
while (True) {
  x = take();
  consume(x);
}
```

- advantages
  - moves all synchronization code into the monitor
  - monitor handles mutual exclusion
  - programmer handles synchronization (buffer full or empty)
  - synchronization is confined to monitor, so it is easier to check for correctness
  - write a correct monitor, any thread can use it (semaphores need to be placed properly in all threads)

Lampson and Redell Monitor

- Hoare monitor requires that signaled thread must run immediately
  - thread that calls `csignal()` must exit the monitor or be suspended
  - for example, when `notempty` condition signaled, thread waiting must be activated immediately or else the condition may no longer be true when it is activated
  - usually restrict `csignal()` to be the last instruction in a method (Concurrent Pascal)
- Lampson and Redell
  - replace `csignal()` with `cnotify()`
  - `cnotify(x)` signals the condition variable, but thread may continue
  - thread at head of condition queue will run at some future time
  - must recheck the condition!
  - used in Mesa, Modula-3

Producer Consumer with a Lampson Redell Monitor

```
char buffer[BUFSIZE];
int in = 0, out = 0, count = 0;
condition notfull, notempty;
```

append:

```
while (count == N)
  cwait(notfull);
buffer[in] = x;
in = (in + 1) % BUFSIZE;
count++;
cnotify(notempty);
```

take:

```
while (count == 0)
  cwait(notempty);
x = buffer[out];
out = (out + 1) % BUFSIZE;
count--;
cnotify(notfull);
```

Lampson Redell Advantages

- allows processes in waiting queue to awaken periodically and reenter monitor, recheck condition
  - prevents starvation
- can also add `cbroadcast(x)`: wake up all processes waiting for condition
  - for example, append variable block of data, consumer consumes variable amount
  - for example, memory manager that frees k bytes, wake all to see who can go with k more bytes
- less prone to error
  - process always checks condition before doing work

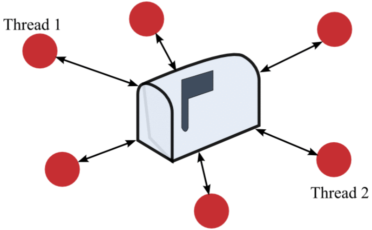
What Can You Do?

- emulate a Lampson Redell Monitor with mutex and condition variables or semaphores
  - create a class with private data only
  - use the same mutex or semaphore to protect all class methods
  - use condition variables or semaphores to replace `cwait()` and `cnotify()`
  - use while loops to recheck conditions
- take your semaphores and move them inside the method call instead of outside of it (see circular buffer implementation)

Message Passing

- needed for distributed systems: no shared memory
- two primitives
  - `send(destination,message)`
  - `recv(source,message)`
- blocking
  - Operating System may block sender or receiver unless you use non-blocking calls
  - when sender returns, this does not mean the message has been delivered, just accepted by the transport protocol

Message Passing with a Mailbox



- create a mailbox abstraction (managed by a separate thread)
  - threads send messages to the central mailbox
  - address message directly to another thread
  - threads poll mailbox to get messages
- provide mutual exclusion
  - use a null message as a token
  - process that gets the token can enter the critical section
  - if token is not available, block until you get it

Producer Consumer with a Mailbox

```
int null = 0;
create_mailbox(mayproduce);
create_mailbox(mayconsume);
for (int i = 1; i<= CAPACITY; i++)
  send(mayproduce, null);
```

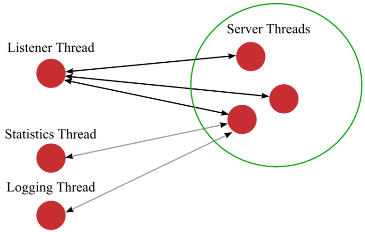
producer:

```
message pmsg;
while (True) {
  receive(mayproduce, pmsg);
  pmsg = produce();
  send(mayconsume, pmsg);
}
```

consumer:

```
while (true) {
    receive(mayconsume, cmsg);
    consume(cmsg);
    send(mayproduce, null);
}
```

Direct Communication



- have a thread in charge of each shared data structure or file
- web server example
  - send a message to request a new connection
  - send a message to log statistics
  - send a message to log a request

Unix Domain Socket Address Structure

```c
struct sockaddr_un {
  sa_family_t      sun_family;      // AF_LOCAL
  char             sun_path[108];   // path name
}
```

- path name must be null terminated

UNIX Domain Server

```
struct sockaddr_un server;
char *filename = "/tmp/mysocket";
bzero(&server, sizeof(server));
server.sin_family = AF_UNIX;
strncpy(server.sun_path,filename,sizeof(server.sun_path) - 1);

s = socket(PF_UNIX,SOCK_STREAM,0)
if (!s) {
  perror("socket");
  exit();
}
if (bind(s,&server,sizeof(server)) < 0) {
  perror("bind");
  exit();
}
```

- call `unlink(filename)` when finished with socket

UNIX Domain Client

```
struct sockaddr_un server;
char *filename = "/tmp/mysocket";
bzero(&server, sizeof(server));
server.sin_family = AF_UNIX;
strncpy(server.sun_path,filename,sizeof(server.sun_path) - 1);

s = socket(PF_UNIX,SOCK_STREAM,0)
if (!s) {
  perror("socket");
  exit();
}
if (connect(s,&server,sizeof(server)) < 0) {
  perror("connect");
  exit();
}
```

Deadlock Definition and Conditions

- *permanent blocking of a set of processes or threads that either compete for system resources or communicate with each other*
- conditions
  1. mutual exclusion: only one thread may use a resource at a time
  2. hold-and-wait: a thread keeps one resource while waiting for another
  3. no preemption: a thread can't be forced to release a resource
  4. circular wait: a cycle of threads waiting for each other
- if first three conditions hold, then deadlock is possible if circular wait occurs
- depends on execution order!

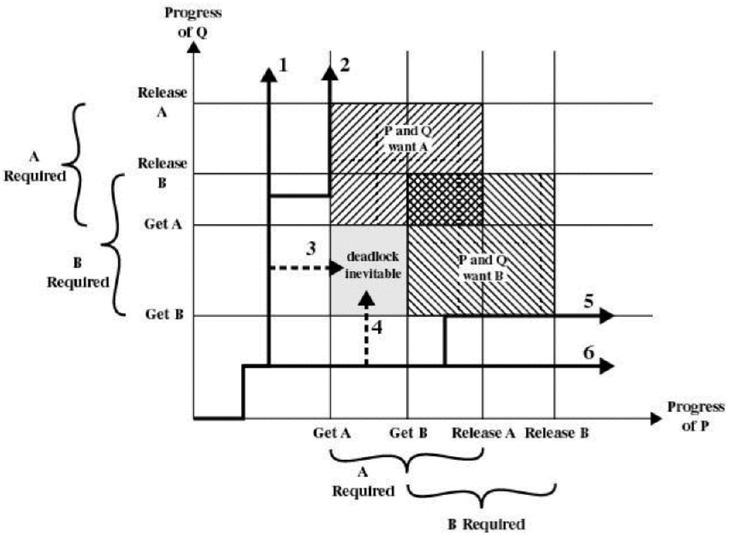
Example

```
Thread P
...
Get A
...
Get B
...
Release A
...
Release B
```

```
Thread Q
...
Get B
...
Get A
...
Release B
...
Release A
```

- *is deadlock possible?*

Deadlock Possibilities



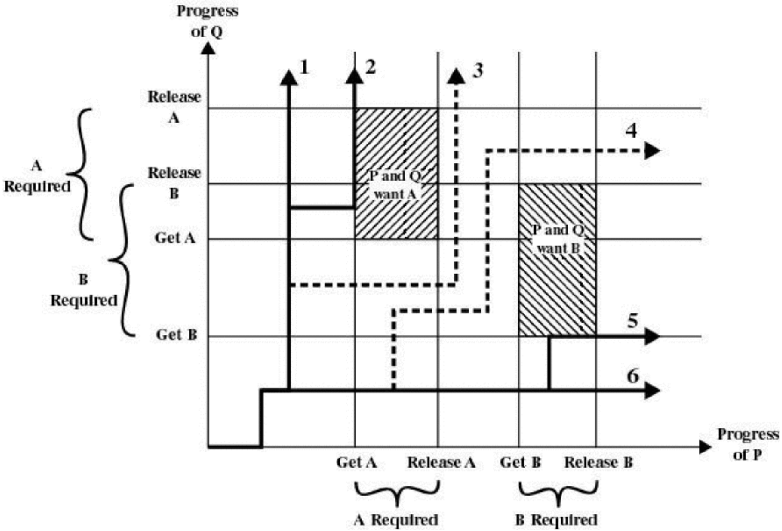
Revised Sample Code

```
Thread P
...
Get A
...
Release A
...
Get B
...
Release B
```

```
Thread Q
...
Get B
...
Get A
...
Release B
...
Release A
```

- *is deadlock possible?*

Deadlock Avoided



Simple Deadlock Prevention

- prevent one of the conditions from happening
- simplest to prevent is hold-and-wait: hold only one resource at a time
- can also prevent circular wait: impose ordering on resources