Synchronization (Part 1)

Learning Objectives

- Understand and solve synchronization problems in multi-threaded and multi-process programs.
- Understand the concepts of race conditions, mutual exclusion, critical section and deadlocks
- Explain and use synchronization primitives such as mutexes and sempahores
- Understand the producer-consumer synchronization design pattern
- Understand how to use basic synchronization primitives in PThreads library and Windows API

https://www.youtube.com/watch?v=k2KMnpD46jI

Interacting Processes/Threads

- Concurrent programs is an umbrella term for multi-threaded programs and multi-process applications.
- Processes (Threads) can be *contending* or *cooperating*. Either way, synchronization is needed.
- *Parallel and Distributed computing is now in the mainstream with multi-core and many-core systems and clusters*.
- Variety of parallel programming languages and systems are available. Most operating systems provide native support for multi-threaded programs and libraries are widely available for parallel programming.

Parallelizing mergesort using threads

- Consider the standard recursive mergesort. It divides the array into two halves, sorts each half recursively and then merges them to sort the entire array. See example code at: mergesort/single-threaded
- ```
  void serial_mergesort(int A[], int p, int r)
  {
      if (r-p+1 <= INSERTION_SORT_CUTOFF)  {
              insertion_sort(A,p,r);
      } else {
          int q = (p+r)/2;
          serial_mergesort(A, p, q);
          serial_mergesort(A, q+1, r);
          merge(A, p, q, r);
      }
  }
  ```
- How would you parallelize mergesort using multiple threads?
  ```

Concurrent Processes

- A concurrent program consists of several sequential processes whose execution sequences are *interleaved*. The sequential processes communicate with each other in order to synchronize or to exchange data.
- Suppose a concurrent process P consists of two processes p_1 and p_2. Then we can imagine *interleaving* as if some supernatural being were to execute the instructions one at a time, each time flipping a coin to decide whether the next one will be from p_1 or p_2. These execution sequences exhaust the possible behaviors of P.
- Suppose p_1 has m instructions and p_2 has n instructions. How many possible interleavings are there?

$$\binom{m+n}{m}$$

(which is exponential!)

Nondeterminism, Race conditions and Critical Sections

- *Race Condition.* When two or more processes are reading or writing shared data and the final result depends on the order in which their instructions get scheduled.
- *Critical Section.* The section of a program where shared data is accessed. We must ensure that of all the processes accessing the same shared data only one process is in its critical section at a time. This is called the *mutual exclusion* problem.
- We want to solve *mutual exclusion* problem without making any assumptions on the speed of the CPU, number of CPUs and the scheduling order of the processes.

An Example of a Race Condition

```
shared double balance; /* shared variable */
/* Code for process p1 */          /* Code schema for process p2 */
...                                 ...
balance = balance + amount          balance = balance - amount
...                                 ...

/* Compiled code for p1 */          /* Compiled code for p2 */
load R1, balance                    load R1, balance
load R2, amount                     load R2, amount
add R1, R2                          sub R1, R2
store R1, balance                   store  R1, balance
```

- Given initial `balance` of \$100, amount to credit to be \$30 and amount to debit to be \$80, how many possible values can `balance` have after the two processes access the shared variable?

- Code example: bad-bank-balance.c Illustrates race conditions when multiple threads access the same global variable and the result depends on the interleaving of threads.

A Model Concurrent Process


```
for (;;) {
    /*pre-protocol*/
    ...
    /*critical section*/
    ...
    /*post-protocol*/
    ...
    /*remainder*/
    ...
}
```

- We assume that the process never terminates in the code for pre-protocol, critical section, post-protocol. A process can terminate abnormally in the remainder section. If a process dies in the remainder section, it should not affect other processes.

Properties of Concurrent Processes

- *Correctness.* Comes in two flavors: safety and liveness.
  - *Safety.* If the program terminates, the answer must be "correct." Safety can always be improved by giving up some concurrency.
  - *Liveness.* If something is supposed to happen, then eventually it will happen. For example:
    - If a process wishes to enter its critical section, then eventually it will do so.
    - If a producer produces data, then eventually the consumer will consume it.

    There are two types of violations of liveness: *deadlock* and *lockout*.
    - *deadlock.* No process is able to make any progress. The absence of deadlock can be shown by proving that there is at least one live process.
    - *lockout* or *starvation.* There is always some process that can make progress but some identifiable process is being indefinitely delayed. This is more difficult to discover and correct.
- *Fairness.* A process wishing to progress must get a fair deal relative to all other processes. No precise definition is possible.

Mutual Exclusion via Disabling Interrupts


```
shared double balance; /* shared variables */

/* Process p1 */                /* Process p2 */

disableInterrupts();            disableInterrupts();
balance = balance + amount;     balance = balance - amount;
enableInterrupts();             enableInterrupts();
```

Mutual Exclusion via Disabling Interrupts

- A simple way to ensure mutual exclusion is to disable interrupts. But disabling interrupts is fraught with pitfalls:
  - User processes may cause havoc like not turning on the interrupts again.
  - What is we have more than one CPU? Doesn't work if the system has more than one CPUs.
  - However, this technique could be useful within the kernel in a limited context. In Linux, this is known as the Big Kernel Lock (BKL).
  - Note: the BKL was removed in Linux version 2.6.39 and replaced with finer grained locking.
  - The git commit that removed the last traces of the BKL is here: http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4ba8216cd90560bc402f52076f64d8546e8aefcb

Synchronization using Shared Memory

- Explicitly synchronize the processes through shared variables. The processes co-operate to ensure mutual exclusion in the critical section. But accessing the shared variables would itself become a critical section.
- *Deadlocks* can also be created while trying to ensure mutual exclusion.

The Rules of the Game

1. A concurrent program will consist of two or more sequential programs whose execution sequences are interleaved.
2. The processes must be loosely connected. In particular, the failure of any process outside its critical section and protocols must not affect other processes.
3. A concurrent program is correct if it does not suffer from violation of safety properties such as mutual exclusion and of liveness properties such as deadlock and lockout.
4. A concurrent program is incorrect *if there exists an interleaved execution sequence which violates a correctness requirement*. Hence it is sufficient to construct a scenario to show incorrectness; to show correctness requires a mathematical argument that the program is correct for all execution sequences.
5. No timing assumptions are made except that no process halts in its critical section and that, if there are ready processes, one is eventually scheduled for execution. We may impose other fairness requirements.
6. We shall assume some primitive synchronization instructions such as a memory arbiter, which guarantees that each memory access is an atomic (indivisible) operation.

The Igloo Metaphor

Only one person can enter the igloo at a time. The small size of the igloo is a metaphor for the memory arbiter. The Igloo example is borrowed from the book *"Principles of concurrent programming"* by M. Ben-Ari.

First Attempt

```
int turn=1; /* shared variable turn */
void  P1() {
    for (;;) {
        while (turn == 2); //do nothing
        /* critical_section_1 */
        turn = 2;
        /* remainder_1 */
    }
}
void  P2() {
    for (;;) {
        while (turn == 1); //do nothing
        /* critical_section_2 */
        turn = 1;
        /* remainder_2 */
    }
}
void main()
{
    thread_t thread1, thread2;

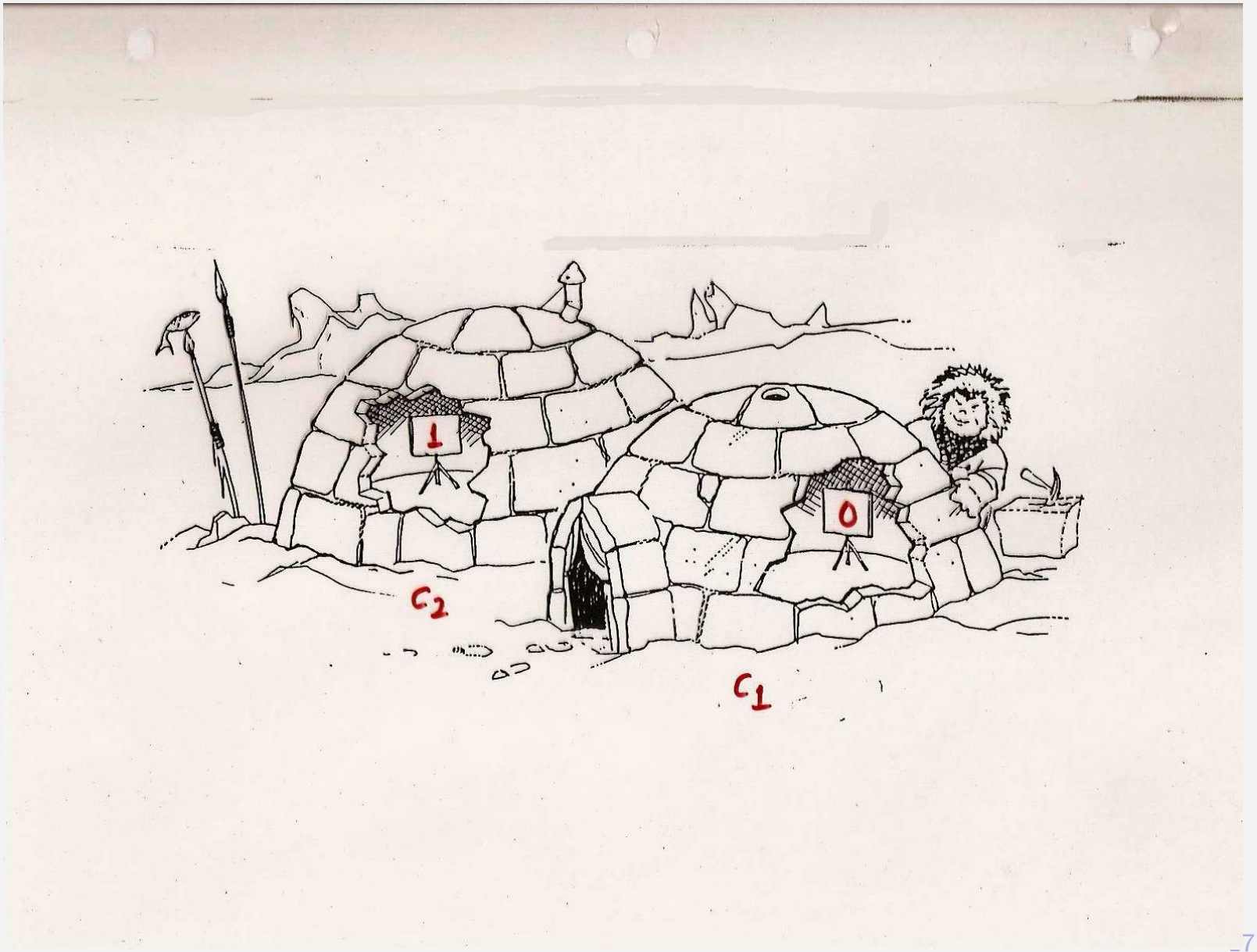
    pthread_create(&thread1, NULL, P1, NULL);
    pthread_create(&thread2, NULL, P2, NULL);
    pthread_join(&thread1, NULL);
    pthread_join(&thread2, NULL);
}
```

First Attempt (contd.)

- Satisfies mutual exclusion, no deadlock or lockout, but not loosely connected (think polar bears!)
- Fairness: One process is forced to work at the pace of the other process.

This technique of passing control explicitly from one process to another is known as *coroutines*.

The Igloo Metaphor (contd.)



```
int c1=FALSE; /* shared variable c1 */
int c2=FALSE; /* shared variable c2 */

void  P1() {
    for (;;) {
        while (c2); //do nothing
        c1 = TRUE;
        /* critical_section_1 */
        c1 = FALSE;
        /* remainder_1 */
    }
}

void  P2() {
    for (;;) {
        while (c1); //do nothing
        c2 = TRUE;
        /* critical_section_2 */
        c2 = FALSE;
        /* remainder_2 */
    }
}
void main() { /* same as before */}
```

Second Attempt (contd.)

- Does not satisfy mutual exclusion.
- Give an example interleaving that shows both processes in the critical section.

Third Attempt

```
int c1=FALSE; /* shared variable c1 */
int c2=FALSE; /* shared variable c2 */
```

```
void  P1() {
    for (;;) {
        c1 = TRUE;
        while (c2); //do nothing
        /* critical_section_1 */
        c1 = FALSE;
        /* remainder_1 */
    }
}
```

```
void  P2() {
    for (;;) {
        c2 = TRUE;
        while (c1); //do nothing
        /* critical_section_2 */
        c2 = FALSE;
        /* remainder_2 */
    }
}
```

```
void main() { /* same as before */}
```

Third Attempt (contd.)

- This solution satisfies mutual exclusion but deadlocks.
- However, it is still instructive to prove that the solution satisfies mutual exclusion.

Fourth Attempt

```
int c1=FALSE; /* shared variable c1 */
int c2=FALSE; /* shared variable c2 */

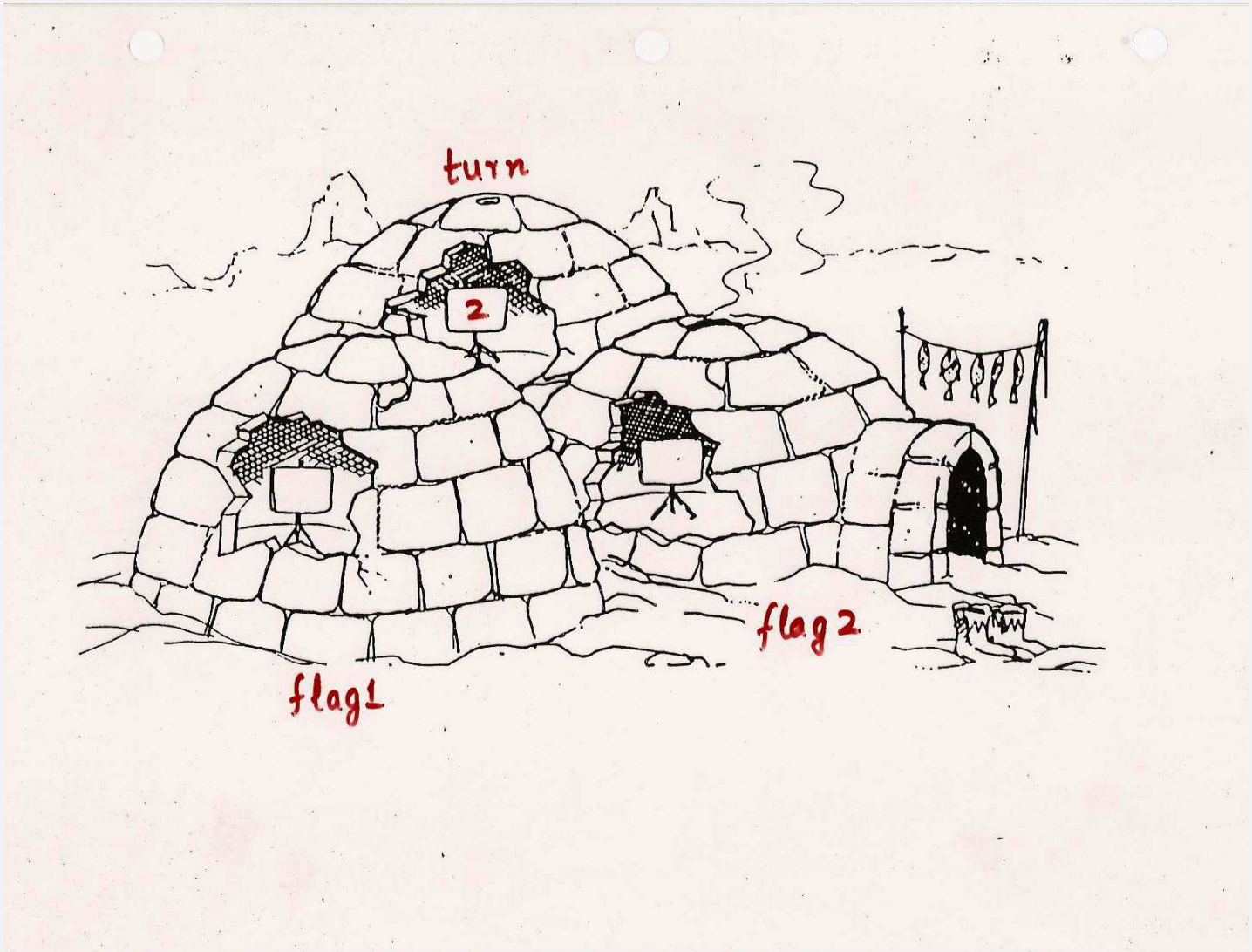
void  P1() {
    for (;;) {
        c1 = TRUE;
        while (c2)  {
            c1 = FALSE;
            //do nothing for
            //a few moments
            c1 = TRUE;
        }
        /* critical_section_1 */
        c1 = FALSE;
        /* remainder_1 */
    }
}

void  P2() {
    for (;;) {
        c2 = TRUE;
        while (c1)  {
            c2 = FALSE;
            // do nothing for a few moments
            c2 = TRUE;
        }
        /* critical_section_2 */
        c2 = FALSE;
        /* remainder_2 */
    }
}
```

Fourth Attempt (contd.)

- Does satisfy mutual exclusion.
- Makes the processes more polite (or chivalrous). However, there can be such a thing as too much chivalry. This creates a potential lockout situation.
- Come up with an example where the two processes lockout due to too much chivalry!

The Igloo Metaphor (contd.)



Petersen's Solution


```
int flag1=FALSE; /* shared variable flag1 */
int flag2=FALSE; /* shared variable flag2 */
int turn=1;      /* shared variable turn */

void  P1() {
    for (;;) {
        flag1 = TRUE;
        turn = 2;
        while (flag2 && (turn == 2)); //do nothing
        /* critical_section_1 */
        flag1 = FALSE;
        /* remainder_1 */
    }
}
void  P2() {
    for (;;) {
        flag2 = TRUE;
        turn = 1;
        while (flag1 && (turn == 1)); //do nothing
        /* critical_section_2 */
        flag2 = FALSE;
        /* remainder_2 */
    }
}
void main() { /* same as before */}
```

Proof for Mutual Exclusion:
The following proof is symmetric and we show the arguments only for one case, i.e., for P_1 in its critical section. We have substituted *flag*1 and *flag*2 by an array *flag*[1..2] for notational convenience.

1. (When P_1 entered its critical section) then ((*flag*[2] = *false*) or (*turn* = 1)) *This is due to the test in the while loop.*
2. (When P_1 entered its critical section) then (*flag*[1] = *true*) *This is true since the critical section is bracketed between assignments to flag*[1].
3. ((*turn* = 1) and (*flag*[1] = *true*)) implies (P_2 cannot enter its critical section) *This follows from the test in the while loop for* P_2.
4. (*flag*[2] = *false*) implies (P_2 is out of its critical section) *flag*[2] *is set to false when* P_2 *leaves its critical section or in the initialization. Thus flag*[2] *being false implies that* P_2 *is out of its critical section and is not intending to enter. If it intends to enter it will set flag*[2] *to true.*
5. (When P_1 entered its critical section) then (P_2 is not in its critical section) *Follows from (3) and (4).*
6. (As long as P_1 is in its critical section) then (P_2 does not enter its critical section) *This follows from (5). Since (5) refers to an arbitrary instant of time, then as long as its antecedent (*P_1 *in* critical_section*) remains true, so will its consequent (*P_2 *does not enter critical_section).*

Proof for no deadlock

A deadlock can occur only if the processes are stuck in their **while** loops.

A process P_i is prevented from entering its critical section only if the condition $flag[j] = true$ and $turn = j$ where j is 1 if i is 2 and vice versa. If P_j has set $flag[j] = true$ and is also executing in its **while** loop, then either $turn = 1$ or $turn = 2$, but cannot be both. Thus either P_1 or P_2 will be able to enter its critical section and there is no deadlock.

Proof for no starvation

As noted in the previous proof, either P_1 or P_2 is able to enter the critical section if both attempt to enter at about the same time. To show that there is no starvation, we must show the following two things:

1. Once P_1 gets in to the critical section then it can't prevent P_2 from getting its chance and vice versa.

   Proof: Once P_1 exits its critical section, it will reset $flag[1] = false$ allowing P_2 a chance to enter its critical section. If P_1 resets $flag[1] = true$ in an attempt to quickly re-enter its critical section, it must also set $turn = 2$ (the *politeness* clause). Thus since P_2 does not change the value of the variable *turn* in the execution of the while statement, P_2 will enter its critical section after at most one entry by P_1. The same argument applies if the role of P_1 and P_2 are interchanged.
2. If P_1 terminates in its remainder section (that is, out of its critical section) then that does not prevent P_2 from entering its critical section (and vice versa).

   Proof: Follows from the fact that after leaving the critical section the process P_1 must set $flag[1] = false$ allowing P_2 to enter its critical section. Note that we assume that processes do not terminate during the critical section as well as during the pre-protocol and post-protocol where they are entering or leaving the critical section but they can die in the remainder section.

The Bakery Problem


```
unsigned int choosing[n];
unsigned int number[n];
// initial values:
//       choosing[i] = FALSE; 0 <= i <= n-1
//       number[i] = 0; 0 <= i <= n-1

void p(int i)
{
  for (;;) {
    choosing[i] = TRUE;
    number[i] = MAX(number[0], number[1], ..., number[n-1])+1;
    choosing[i] = FALSE;

    for (j=0; j<n; j++) {
        while (choosing[j]); //do nothing
        // (a,b) < (c,d) if a < c or if a==c and b < d
        while ((number[j] != 0) && ((number[j],j) < (number[i],i)));
    }
    /* critical section */
    ...
    number[i] = 0;
    /* remainder */
  }
}
```

Does the above satisfy mutual exclusion? Does it prevent deadlock or lockout?
Can the numbers overflow? Check the example counting.c

Semaphores

A *semaphore s* is a non-negative integer variable. Once *s* has been given its initial value, the only permissible operations are:

P(s) (or *wait(s)* or down(s)). [if (s $>$ 0) then s = s-1; else {(wait until s $>$ 0)}] If $s > 0$, it is tested and decremented as an atomic operation. However, if *s* is zero, the process executing *P* can be interrupted when it executes the `wait` command.

V(s) (or *signal(s)* or up(s) or post(s)). [s = s+1]. Increment *s* as an indivisible operation. The effect is to signal some process that is blocked on the semaphore.

Semaphores (contd.)

- Semaphores can be *binary*, that is, taking only values 0 or 1. A binary semaphores is often referred to as a Mutex.
- Or they can be *counting* or *general* semaphores, which can take on any value greater than or equal to zero.
- If more than one process is blocked on a semaphore s, than an arbitrary one of these process is woken up by the V(s) operation.
- P stands for *proberen* (to probe) and V stands for *verhogen* (to increment) in Dutch. Semaphores were introduced by E.W. Dijkstra.

The igloo metaphor: Now the igloo not only has a blackboard but also a deep-freezer.

A Train Semaphore

Critical Section Using a Semaphore


```
semaphore mutex = 1;  //must be created and initialized in main()
```

```
process0()
{
  while (TRUE) {
    <compute section>
    wait(mutex);
       <critical section>
    signal(mutex);
  }
}
```

```
process1()
{
  while (TRUE) {
    <compute section>
    wait(mutex);
       <critical section>
    signal(mutex);
  }
}
```

Bank Account example using a semaphore


```
semaphore mutex = 1;  // must be created and initialized in main()
pthread_create(thread1, 0);
pthread_create(thread2, 0);
...
/* thread1 and thread2 call credit/debit multiple times */
...
```

```
credit() {
    /* Enter critical section */
    wait(mutex);
      balance = balance + amount;
    /* Exit critical section */
    signal(mutex);
}
```

```
debit() {
     /* Enter critical section */
      wait(mutex);
         balance = balance - amount;
     /* Exit critical section */
      signal(mutex);
}
```

Interacting Parallel Processes


```
Shared double x,y;    // must be created and setup in main()
```

```
processA()
{
  while (TRUE) {
    <compute A1>;
    write(x); /* produce x */
    <compute A2>;
    read(y); /* consume y */
  }
}
```

```
processB()
{
  while (TRUE) {
      read(x); /* consume x */
      <compute B1>;
      write(y); /* produce y */
      <compute B2>;
  }
}
```

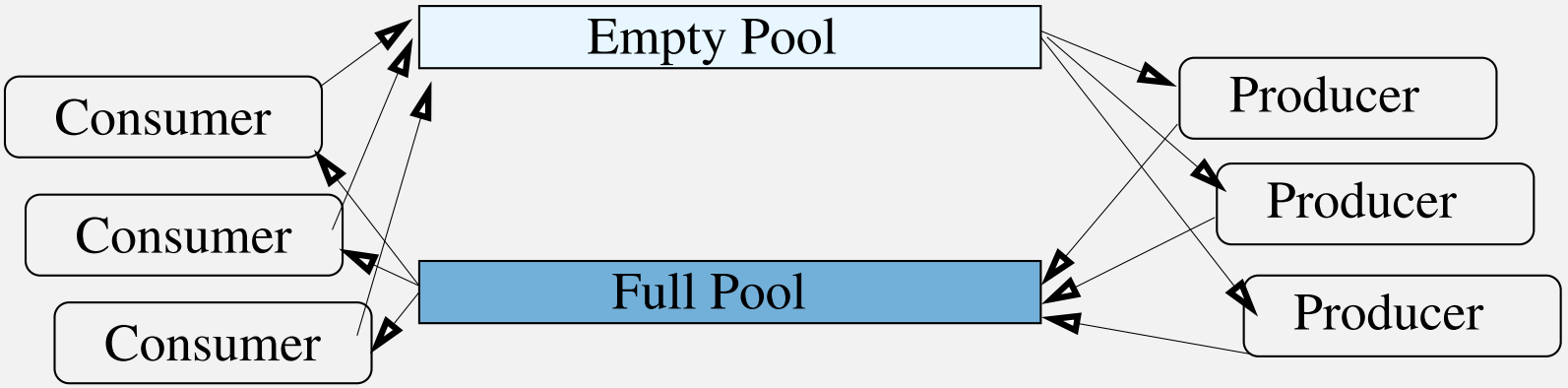
Synchronizing Processes


```
Shared double x,y;   // must be created and setup in main()
semaphore s_x = 0, s_y = 0;
```

```
processA()
{
  while (TRUE) {
    <compute A1>;
    write(x); /* produce x */
    signal(s_x); /* signal B */
    <compute A2>;
    /* Wait for signal from B */
    wait(s_y);
    read(y); /* consume y */
  }
}
```

```
processB()
{
  while (TRUE) {
    /* Wait for signal from A */
    wait(s_x);
    read(x); /* consume x */
    <compute B1>;
    write(y); /* produce y */
    signal(s_y); /* signal A */
    <compute B2>;
  }
}
```

Producers and Consumers



Producers and Consumers


```
producer() {
  buf_type *next, *here;
  while(TRUE) {
    produce_item(next);
    /*Claim an empty buffer*/
    wait(empty);
    wait(mutex);
    here = obtain(empty);
    signal(mutex);
    copy_buffer(next, here);
    wait(mutex);
    release(here, fullPool);
    signal(mutex);
    /*Signal a full buffer*/
    signal(full);
  }
}

semaphore mutex = 1;
/* counting semaphores */
semaphore full = 0;
semaphore empty = N;
buf_type buffer[N];
pthread_create(producer, 0);
pthread_create(consumer, 0);
```

```
consumer() {
  buf_type *next, *here;
  while(TRUE) {
    /*Claim full buffer*/
    wait(full);
    /*Manipulate the pool*/
    wait(mutex);
    here = obtain(full);
    signal(mutex);
    copy_buffer(here, next);
    /*Manipulate the pool*/
    wait(mutex);
    release(here, emptyPool);
    signal(mutex);
    /*Signal an empty buffer*/
    signal(empty);
    consume_item(next);
  }
}
```

More on Producers and Consumers

- What happens if we interchange the wait(full) and wait(mutex) operations? (in the consumer)
- What happens if we interchange the signal(full) and signal(mutex) operations? (in the consumer)
- *How to improve the performance while retaining correctness?*
  - Separate semaphores for full/empty pools?
  - Multiple queues?
  - For multiple queues, should we let producers and consumers access a queue at random or should there be a systematic pattern of access?

Synchronization in POSIX Threads (PThreads)

Mutexes are simple lock primitives that can be used to control access to a shared resource.

```
#include<pthread.h>
pthread_mutex_t <variable>;
pthread_mutex_init(pthread_mutex_t *, pthread_mutexattr_t *)
pthread_mutex_lock(pthread_mutex_t *)
pthread_mutex_trylock(pthread_mutex_t *)
pthread_mutex_unlock(pthread_mutex_t *)
pthread_mutex_destroy(pthread_mutex_t *)
```

semaphores in POSIX threads support the following operations.

```
#include<pthread.h>
#include <semaphore.h>
sem_t <variable>;
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);
int sem_getvalue(sem_t *sem, int *sval);
int sem_destroy(sem_t *sem);
```

PThreads Synchronization

- Mutex - A construct used to protect access to a shared bit of memory.
- Think of a lock that only has one key. If you want to open the lock you must get the key. If you don't have the key you must wait until it becomes available.
- A Mutex, short for *Mutual exclusion object*, is an object that allows multiple program threads to share the same resource, such as a data structure or file access, but not simultaneously. Each thread locks the mutex to gain access to the shared resource and then unlocks when it is done. We can use mutexes to prevent race conditions.

Mutual Exclusion Using Locks


```
pthread_mutex_t mutex;
void  P1() {
    for (;;) {
        pthread_mutex_lock( &mutex );
        /* critical_section_1 */
        pthread_mutex_unlock( &mutex );
        /* remainder_1 */
    }
}
void  P2() {
    for (;;) {
        pthread_mutex_lock( &mutex );
        /* critical_section_2 */
        pthread_mutex_unlock( &mutex );
        /* remainder_2 */
    }
}
void main()
{
    thread_t thread1, thread2;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(thread1, NULL, P1, NULL);
    pthread_create(thread2, NULL, P2, NULL);
    pause(); // let the threads play forever
}
```

Semaphores in Pthreads


```
#include <semaphore.h>
```

`int sem_init(sem_t *sem, int pshared, unsigned int value);` Initializes the semaphore object pointed to by sem. The count associated with the semaphore is set initially to value. The flag `pshared` should be set to zero. Non-zero value allows semaphores to be shared across processes.

`int sem_wait(sem_t *sem);` Suspends the calling thread until the semaphore pointed to by sem has non-zero count. It then atomically decreases the semaphore count.

```
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

`int sem_trywait(sem_t *sem);` A non-blocking variant of `sem_wait`

`int sem_post(sem_t *sem);` Atomically increases the count of the semaphore pointed to by sem. This function never blocks and can safely be used in asynchronous signal handlers.

```
int sem_getvalue(sem_t *sem, int *sval);
```

```
int sem_destroy(sem_t *sem);
```

PThread Synchronization Examples

- See the example safe-bank-balance.c for a solution to the race condition using a Mutex lock.
- Sychronized hello world: threads-hello-synchronized.c
- File copy using two threads (reader and writer): threads-sem-cp.c
- File copy using double buffering: threads-dbl-buf.c

In-class Exercise (1)

Dining Philosophers? Dining Semaphores? We have 5 philosophers that sit around a table. There are 5 bowls of rather entangled spaghetti that they can eat if they get hungry. There are five forks on the table as well. However, each philosopher needs two forks to eat the tangled spaghetti. No two philosophers can grab the same fork at the same time. We want the philosophers to be able to eat amicably. Consider the following solution to this problem.

```
/* dining_philosophers */
sem_t fork[5]; // array of binary semaphores
sem_t table; // general semaphore

void philosopher(void *arg)
{
    i = *(int *) arg;
    for (;;) {
        think();
        sem_wait(&table);
        sem_wait(&fork[i]);
        sem_wait(&fork[(i+1) % 5];
        eat();
        sem_post(&fork[i]);
        sem_post(&fork[(i+1) % 5];
        sem_post(&table);
    }
}
```

In-class Exercise (2)


```
. . .
int main()
{
    int i;
    for (i = 0; i< 5; i++) {
        sem_init(&fork[i], 0, 1); //initialize to 1
    }
    sem_init(&table, 0, 4); //initialize to 4
    for (i = 0; i< 5; i++) {
        pthread_create(&tid[i], NULL, philosopher, (void *)&i);
    }
    for (i = 0; i< 5; i++) {
        pthread_join(tid[i], NULL);
    }
    exit(0);
}
```

- Argue why it is not possible for more than one philosopher to grab the same fork?
- Can the philosophers ever deadlock. Explain.
- Can a philosopher starve?
- What may happen if we initialize the table semaphore to 5 (instead of 4)?

Other Useful Thread Functions

- `pthread_yield()` Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.
- `pthread_t me = pthread_self()` Allows a pthread to obtain its own identifier
- `pthread_detach(thread)` Informs the library that the threads exit status will not be needed by subsequent `pthread_join` calls resulting in better threads performance.
- Barriers (Not available in Mac OS X)

```
pthread_barrier_t barrier;
pthread_barrier_init(&barrier, NULL, count);
result = pthread_barrier_wait(&barrier);
/* One thread gets PTHREAD_BARRIER_SERIAL_THREAD back
   while others get a zero */
pthread_barrier_destroy(&barrier);
```

See the example threads-barrier.c.

Further Information on POSIX Threads

- Where can I find out more about Threads? On Linux, try
  man -k pthread
  to see the man pages for Pthreads (pthreads) package.
- Check out the following books:
  - Lewis and Berg: *Multithreaded Programming with Pthreads* (Prentice Hall)

Synchronization in MS Windows API

MS Windows API supports Mutex and semaphore objects.

- The methods for Mutexes include `CreateMutex(..)`, `WaitForSingleObject(...)` to wait for it and `ReleaseMutex(...)` to release the Mutex.
- The methods for semaphores include `CreateSemaphore(..)`, `WaitForSingleObject(...)` to wait for it and `ReleaseSemaphore(...)` to release the semaphore.
- A `WaitForMultipleObjects(..)` call is also provided.

Threads in MS Windows API

Get detailed information from `http://msdn.microsoft.com/library/`.

```
HANDLE WINAPI CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);

DWORD WINAPI ThreadProc(
  LPVOID lpParameter
);
```

Semaphores and Mutexes in MS Windows API


```
HANDLE WINAPI CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG lInitialCount,
  LONG lMaximumCount,
  LPCTSTR lpName
);

BOOL WINAPI ReleaseSemaphore(
  HANDLE hSemaphore,
  LONG lReleaseCount,
  LPLONG lpPreviousCount
);

HANDLE WINAPI CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,
  LPCTSTR lpName
);

BOOL WINAPI ReleaseMutex(HANDLE hMutex);
```

Wait calls in MS Windows API


```
DWORD WINAPI WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds
);

DWORD WINAPI WaitForMultipleObjects(
  DWORD nCount,
  const HANDLE* lpHandles,
  BOOL bWaitAll,
  DWORD dwMilliseconds
);
```

Critical Section call in MS Windows API


```
CRITICAL_SECTION cs;
InitializeCriticalSection(&cs);
EnterCriticalSection(&cs);
LeaveCriticalSection(&cs);
```

Multithreaded Example in MS Windows API

- Code Example: thread-sem-cp.c

Synchronization in Java

- Java has the `synchronized` keyword for guaranteeing mutually exclusive access to a method or a block of code. Only one thread can be active among all synchronized methods and synchronized blocks of code in a class.
- Java provides a mutex/lock implementation via the class `ReentrantLock` in the `java.util.concurrent.lock` package.
- Java provides sempahore implementation via the class `Semaphore` in the `java.util.concurrent` package.
- Java synchronization will be covered in the next chapter as it is based on the concept of a Monitor, which is covered in the next chapter.