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.
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. How would you parallelize mergesort?

Sort two halves in parallel using threads and then merge them. However if we sort the two halves and attempt to perform the merging also in parallel, we will run into problems....
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!)
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 schema for process p1
...
balance = balance + amount
...

Code schema for process p2
...
balance = balance - amount
...

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

/* Compiled code for p2 */
load R1, balance
load R2, amount
sub R1, R2
store R1, balance
/* synchronization-part1/bad-bank-balance.c */
/* appropriate header files */
typedef struct account account;
struct account {
    double balance;
};
account *myacct;

void *threadMain(void *);
pthread_t *tids;
int numThreads;
int count;

int main(int argc, char **argv)
{
    int i;
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <numThreads> <iterations>\n", argv[0]);
        exit(1);
    }
    numThreads = atoi(argv[1]);
    count = atoi(argv[2]);
    if (numThreads > 32) {
        fprintf(stderr, "Usage: %s Too many threads specified. Defaulting to 32.\n", argv[0]);
        numThreads = 32;
    }
    myacct = (account *) malloc(sizeof(account));
    myacct->balance = 0.0;
    printf("initial balance = %lf\n", myacct->balance);
Bad Bank Balance Example II

tids = (pthread_t *) malloc(sizeof(pthread_t)*numThreads);
for (i=0; i<numThreads; i++)
    pthread_create(&tids[i], NULL, threadMain, (void *) NULL);

for (i=0; i<numThreads; i++)
    pthread_join(tids[i], NULL);

printf("final balance = %lf\n", myacct->balance);
exit(0);

}

void *threadMain(void *arg)
{
    int i;
    int amount;

    for (i=0; i<count; i++) {
        amount = 1;
        myacct->balance += amount;
    }
    pthread_exit(NULL);
}
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 */

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

/* Process p2 */

disableInterrupts();
balance = balance - amount;
enableInterrupts();
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 is 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
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.
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.)
Second Attempt

```c
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

```c
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.
Proof of mutual exclusion

Even though this attempt deadlocks, it does satisfy the mutual exclusion property. By symmetry it is sufficient to show that: \((P_1 \text{ in } \text{critical\_section\_1}) \text{ implies that } (P_2 \text{ is not in } \text{critical\_section\_2})\).

1. (When \(P_1\) entered \text{critical\_section\_1}) then \((c_2 \text{ was not true})\).
   This follows from the structure of the program, namely the test on \(c_2\) by \(P_1\).

2. (\(c_2\) is not true) implies \((P_2 \text{ is not in } \text{critical\_section\_2})\).
   \text{critical\_section\_2} is bracketed between assignments to \(c_2\) which ensure that this statement is always true.

3. (When \(P_1\) entered \text{critical\_section\_1}) then \((P_2 \text{ is not in } \text{critical\_section\_2})\).
   This is a logical consequence of (1) and (2).

4. \((P_1 \text{ in } \text{critical\_section\_1}) \text{ implies } (c_1 \text{ is true})\).
   \text{critical\_section\_1} is bracketed between assignments to \(c_1\).

5. (\(c_1\) is true) implies \((P_2 \text{ does not enter } \text{critical\_section\_2})\).
   The test will not allow \(P_2\) through.

6. \((P_1 \text{ in } \text{critical\_section\_1}) \text{ implies } (P_2 \text{ does not enter } \text{critical\_section\_2})\).
   This is a logical consequence of (4) and (5).

7. As long as \((P_1 \text{ is in } \text{critical\_section\_1}), (P_2 \text{ will never enter } \text{critical\_section\_2})\).
   This follows from (6). Since (6) refers to an arbitrary instant of time, then as long as its antecedent \((P_1 \text{ in } \text{critical\_section\_1})\) remains true, so will its consequent \((P_2 \text{ does not enter } \text{critical\_section\_2})\).

8. \((P_1 \text{ in } \text{critical\_section\_1}) \text{ implies that } (P_2 \text{ is not in } \text{critical\_section\_2})\).
   Follows from (3) and (7).
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 */
    }
}
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.
The Igloo Metaphor (contd.)
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 \( \text{flag}1 \) and \( \text{flag}2 \) by an array \( \text{flag}[1..2] \) for notational convenience.

1. (When \( P_1 \) entered its critical section) then \(( \text{flag}[2] = \text{false} ) \) or \(( \text{turn} = 1 ) \)
   This is due to the test in the while loop.

2. (When \( P_1 \) entered its critical section) then \(( \text{flag}[1] = \text{true} ) \)
   This is true since the critical section is bracketed between assignments to \( \text{flag}[1] \).

3. \(( \text{turn} = 1 ) \) and \(( \text{flag}[1] = \text{true} ) \) implies \(( P_2 \) cannot enter its critical section) 
   This follows from the test in the while loop for \( P_2 \).

4. \(( \text{flag}[2] = \text{false} ) \) implies \(( P_2 \) is out of its critical section) 
   \( \text{flag}[2] \) is set to \text{false} when \( P_2 \) leaves its critical section or in the initialization. Thus \( \text{flag}[2] \) being \text{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 \( \text{flag}[2] \) to \text{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).
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 $\text{flag}[j] = true$ and $\text{turn} = j$ where $j$ is 1 if $i$ is 2 and vice versa. If $P_j$ has set $\text{flag}[j] = true$ and is also executing in its *while* loop, then either $\text{turn} = 1$ or $\text{turn} = 2$, 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

```c
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 lab/synchronization-part1/counting.c
A *semaphore* \( s \) is a non-negative integer variable. Once \( s \) has been given its initial value, the only permissible operations are:

\[ P(s) \text{ (or } \text{wait}(s) \text{ or } \text{down}(s)) \text{. } \begin{cases} \text{if } (s > 0) \text{ then } s = s-1; \text{ else} \\ \{\text{(wait until } s > 0)\} \end{cases} \]

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.

\[ V(s) \text{ (or } \text{signal}(s) \text{ or } \text{up}(s) \text{ or } \text{post}(s)) \text{. } [s = s+1]. \]

Increment \( s \) as an indivisible operation. The effect is to signal some process that is blocked on the semaphore.
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.
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);
    }
}
Semaphore mutex = 1; // must be created and initialized in main()

process0()
{
  while (TRUE) {
    ... /* Enter critical section */
    wait(mutex);
    balance = balance + amount;
    /* Exit critical section */
    signal(mutex);
    ...
  }
}

process1()
{
  while (TRUE) {
    ... /* 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 s1 = 0, s2 = 0;

processA()
{
    while (TRUE) {
        <compute A1>;
        write(x); /* produce x */
        signal(s1); /* signal B */
        <compute A2>;
        /* Wait for signal from B */
        wait(s2);
        read(y); /* consume y */
    }
}

processB()
{
    while (TRUE) {
        /* Wait for signal from A */
        wait(s1);
        read(x); /* consume x */
        <compute B1>;
        /* Wait for signal from B */
        write(y); /* produce y */
        signal(s2); /* 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?
Implementing Semaphores with Interrupts

class Semaphore {
    int value;
    public:
        Semaphore (int v = 1) {
            // allocate space for the semaphore object in the OS
            value = v;
        }

        wait() {
            disableInterrupts();
            // loop until value is positive
            while (value == 0) {
                enableInterrupts(); // let interrupts occur for a short period
                disableInterrupts();
            }
            value--;
            enableInterrupts();
        }

        signal() {
            disableInterrupts();
            value++;
            enableInterrupts();
        }
    }
}
Implementing a Binary Semaphore with Test-And-Set

```plaintext
boolean s = FALSE;
...
while (TS(s)) ;
  <critical section>
  s = FALSE;
  ...
  <critical section>
Semaphore s = 1;
...
wait(s);
  <critical section>
signal(s);
  ...
```

A test-and-set instruction is an instruction used to write to a memory location and return its old value as a single non-interruptible (atomic) operation.

Test-And-Set is an example of an atomic operation. Atomic operations are architecture-dependent. Using these we can create spinlocks to solve mutual exclusion.
Implementing Counting Semaphore with Test-And-Set

```c
struct semaphore {
    int value = <initial value>;
    boolean mutex = FALSE;
    boolean hold = TRUE;
};

shared struct semaphore s;

wait(struct semaphore s) {
    while (TS(s.mutex));
    s.value = s.value - 1;
    if (s.value < 0) {
        s.mutex = FALSE;
        while (!s.hold);
    } else {
        s.mutex = FALSE;
    }
}

signal(struct semaphore s) {
    while (TS(s.mutex));
    s.value = s.value + 1;
    if (s.value <= 0) {
        while (!s.hold);
        s.hold = FALSE;
    } else {
        s.mutex = FALSE;
    }
}
```
The statement: while (!s.hold); is needed because otherwise we can have the following race condition: A process believes it is blocked in the wait procedure, yet the signal procedure encounters s.hold as being TRUE. This situation can occur when consecutive signal operations occur before any process executes a wait operation. Without the while statement, the result of one of the signal operations would be lost.

The busy waiting can be avoided by putting a yield() call in the body of the while loops.

The above implementation isn’t a full semaphore. For that we need to maintain a queue where we put waiting processes to sleep instead of busy-waiting.
Mutexes are simple lock primitives that can be used to control access to a shared resource.

```c
#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.

```c
#include<pthread.h>
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t * sem);
in sem_trywait(sem_t * sem);
in sem_post(sem_t * sem);
in sem_getvalue(sem_t * sem, int * sval);
in sem_destroy(sem_t * sem);
```
Mutual Exclusion Using Locks

```c
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. It is reserved for future use.

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);
Safe Bank Balance Example

/* synchronization-part1/safe-bank-balance.c */
/* appropriate header files */
typedef struct account account;
struct account {
    double balance;
    pthread_mutex_t mutex;
};
account *myacct;

void *threadMain(void *);
pthread_t *tids;
int numThreads;
int count;
int main(int argc, char **argv)
{
    int i;
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <numThreads> <iterations>\n", argv[0]);
        exit(1);
    }
    numThreads = atoi(argv[1]);
    count = atoi(argv[2]);
    if (numThreads > 32) {
        fprintf(stderr, "Usage: %s Too many threads specified. Defaulting to 32.\n", argv[0]);
        numThreads = 32;
    }
    myacct = (account *) malloc(sizeof(account));
    myacct->balance = 0.0;
    pthread_mutex_init(&(myacct->mutex), NULL);
    printf("initial balance = %lf\n", myacct->balance);

    tids = (pthread_t *) malloc(sizeof(pthread_t)*numThreads);
    for (i=0; i<numThreads; i++)
        pthread_create(&tids[i], NULL, threadMain, (void *) NULL);

    for (i=0; i<numThreads; i++)
        pthread_join(tids[i], NULL);
    printf("final balance = %lf\n", myacct->balance);
    exit(0);
}
void *threadMain(void *arg) {
    int i;
    int amount;

    for (i=0; i<count; i++) {
        amount = 1;
        pthread_mutex_lock(&(myacct->mutex));
        myacct->balance += amount;
        pthread_mutex_unlock(&(myacct->mutex));
    }
    pthread_exit(NULL);
}
Synchronized Hello World

/* include files, prototypes etc. */
sem_t worlds_turn;
main() {
    pthread_t thread1, thread2;
    char *message1 = "Hello";
    char *message2 = "World\n";

    sem_init(&worlds_turn, 0, 1);
    sem_wait(&worlds_turn); /* world goes second */

    pthread_create(&thread1, NULL, print_message_function, (void *) message1);
    sem_wait(&worlds_turn);
    pthread_create(&thread2, NULL, print_message_function, (void *) message2);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    sem_destroy(&worlds_turn);
    printf("\n");
    exit(0);
}

...
Synchronized Hello World

...  
void print_message_function(void *ptr)  
{  
  char *message;  
  message = (char *) ptr;  
  printf("%s ", message);  
  fflush(stdout);  
  sem_post(&worlds_turn);  
  sem_post(&child_counter);  
  pthread_exit(0);  
}
/* include files etc. */
int in, out;
int src, dst;
int bufsize;
char buf[MAX_BUF_SIZE];
sem_t writers_turn;
sem_t readers_turn;
void main(int argc, char *argv[]) {
    pthread_t thread1, thread2;
    ...  // initial stuff
    src = open(argv[2], O_RDONLY);
    if (src < 0) exit(2);
    dst = creat(argv[3], MODE);
    if (dst < 0) exit(3);

    sem_init(&readers_turn, 0, 1);
    sem_init(&writers_turn, 0, 1);
    sem_wait( &writers_turn );  // reader goes first
    pthread_create(&thread1, NULL, (void*)&reader, (void*) NULL);
    pthread_create(&thread2, NULL, (void*)&writer, (void*) NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    close(src); close(dst);
    exit(0);
}
void reader(void)
{
    while (1) {
        sem_wait (&readers_turn);
        in = read(src, buf, bufsize);
        sem_post (&writers_turn);
        if (in <= 0) break;
    }
    pthread_exit(0);
}

void writer(void)
{
    while (1) {
        sem_down (&writers_turn);
        out = write(dst, buf, in);
        sem_post (&readers_turn);
        if (out <= 0) break;
    }
    pthread_exit(0);
}
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)**
  
  ```c
  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);
  ```

**Further Examples**
Also see the examples *threads-dbl-buf.c*, *threads-id.c*, *threads-barrier.c* in the folder *lab/synchronization-part1/*
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)
  - Lewis and Berg: *Multithreaded Programming with Java Technology* (Prentice Hall)
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


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

DWORD WINAPI WaitForMultipleObjects(
    DWORD nCount,
    const HANDLE* lpHandles,
    BOOL bWaitAll,
    DWORD dwMilliseconds
);
CRITICAL_SECTION cs;
InitializeCriticalSection(&cs);
EnterCriticalSection(&cs);
LeaveCriticalSection(&cs);
Multithreaded Example in MS Windows API

/* lab/ms-windows/synchronization-part1/threads-sem-cp.c */
* appropriate header files */
#define MAX_BUF_SIZE 65536
size_t in, out;
FILE *src, *dst;
size_t bufsize;
LONG lPrevCount;
char buf[MAX_BUF_SIZE];

DWORD WINAPI reader(LPVOID resrvd); 
DWORD WINAPI writer(LPVOID resrvd); 
HANDLE hReadersTurn; 
HANDLE hWritersTurn;

int main(int argc, char *argv[]) 
{
HANDLE thread1, thread2;
DWORD thread1ID, thread2ID;

if (argc != 4) {
    fprintf(stderr,"Usage: %s <buffer size> <src> <dest>
", argv[0]);
    exit(1);
}
bufsize = atoi(argv[1]);
if (bufsize > MAX_BUF_SIZE) {
    fprintf(stderr,"Error: %s: max. buffer size is %d
","argv[0], MAX_BUF_SIZE);
    exit(1);
}
src = fopen(argv[2], "r");
if (src < 0) exit(2);
dst = fopen(argv[3], "w");
if (dst < 0) exit(3);
/* ... } */

Multithreaded Example in MS Windows API (contd.)

```c
/* ... */
hReadersTurn = CreateSemaphore(
    NULL,      // Security attributes
    1,         // Initial count
    1,         // Maximum count (binary sem)
    "ReaderSem" // Name of sem
);

hWritersTurn = CreateSemaphore(
    NULL,      // Security attributes
    0,         // Initial count
    1,         // Maximum count (binary sem)
    "WriterSem" // Name of sem
);

/* ... } */
```
/* { ... */
/* create the reader and writer threads */
thread1 = CreateThread(
    NULL, // Security attributes
    0, // Stack size (default)
    &reader, // Start function
    NULL, // Thread Parameter
    0, // Creation flags
    &thread1ID // Thread ID
);

thread2 = CreateThread(
    NULL, // Security attributes
    0, // Stack size (default)
    &writer, // Start function
    NULL, // Thread Parameter
    0, // Creation flags
    &thread2ID // Thread ID
);

/* the main program (i.e. the heavyweight parent process) waits for the threads to finish. */
WaitForSingleObject(thread1, INFINITE);
WaitForSingleObject(thread2, INFINITE);

fclose(src);
fclose(dst);
exit(0);
/* reader(): The function reader() is the entire reader thread. */
DWORD WINAPI reader(LPVOID resrvd)
{
    while (1) {
        WaitForSingleObject(hReadersTurn, INFINITE);
        in = fread(buf, 1, bufsize, src);
        ReleaseSemaphore(hWritersTurn, 1, &lPrevCount);
        if (in <= 0) break;
    }
    ExitThread(0);
}

/* writer(): The main function for the writer thread. */
DWORD WINAPI writer(LPVOID resrvd)
{
    while (1) {
        WaitForSingleObject(hWritersTurn, INFINITE);
        out = fwrite(buf, 1, in, dst);
        ReleaseSemaphore(hReadersTurn, 1, &lPrevCount);
        if (out <= 0) break;
    }
    ExitThread(0);
}
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 synchronization will be covered in the next Chapter in more detail as it is based on the concept of a Monitor, which is covered in the next Chapter.