Synchronization (Part 2)
A **monitor** is designed to allow concurrency while retaining the advantage of a structured construct.

- Each monitor is entrusted with a specific task and in turn it will have its own privileged data and instructions.
- Entry to the monitor by one process excludes entry by any other process.
- Since all processing is encapsulated inside the monitor, the programmer cannot accidentally change the monitor variables.
- A system would be more robust and efficient if we have a monitor for each specific task that involves multiple processes.
Monitors include condition variables to allow processes to signal each other. A condition variable is a global structure inside the monitor that supports three operations.

- **wait()**:Suspends the invoking process until it receives another signal from another process.
- **signal()**:Resumes exactly one process if any processes are waiting on the condition variable. If no process is waiting, then the signal has no effect.
- **queue()**:Returns TRUE if there is at least one process suspended on the condition variable, and FALSE otherwise.
Signal takes effect immediately (Hoare Semantics).

- The signal statement should always be the last statement before a process leaves the monitor. This allows the signaled process to be woken up immediately. So a process that gets past the `wait()` can be guaranteed that the condition is true.
- In case we have processes waiting to enter the monitor as well as processes waiting on a condition variable, then the next process to be allowed in the monitor is taken from the queue of processes waiting on the condition variable.
- The processes waiting in the queue on a condition variable can be woken by the scheduler in either arbitrary, first-come-first-served, or priority order. In the priority scheduling case we will modify the `wait()` to have an argument `wait(priority)`, where priority is an integer with smaller integers implying higher priority.

Signal does not have effect immediately (Hansen Semantics).
The signal is recorded and delivered after the signaling process leaves the monitor. In this case a process on waking up has to test the condition again before proceeding. This would lead to fewer context switches.
Bank Account Monitor

```cpp
monitor sharedBalance
{
    int balance;
    public:
        credit(int amount) { balance = balance + amount;}
        debit(int amount) { balance = balance - amount;}
}
```
Simulation of a Binary Semaphore by a Monitor

monitor SemaphoreSimulation {
    boolean busy = FALSE;
    condition notbusy;
    public:
        down() {
            if (busy) notbusy.wait();
            busy = TRUE;
        }

        up() {
            busy = FALSE;
            notbusy.signal();
        }
}


Readers and Writers

Suppose a resource is to be shared among a community of processes of two types: *readers* and *writers*.

- A *reader* process can share the resource with any other reader process, but not with any writer process.
- A *writer* process requires exclusive access to the resource whenever it acquires any access to the resource.
- Several different policies can be implemented for managing the shared resource.

- *Policy 1: Readers preference.* As long as a reader holds the resource and there are new readers arriving, any writer must wait for the resource to become available. Writer can starve.
- *Policy 2: Writers preference.* When a writer process requests access to the shared resource, any subsequent reader process must wait for the writer to gain access to the shared resource and then release it. Reader can starve.
- *Policy 3: Give neither preference.* Neither readers nor writers can starve.
monitor ReadersWriters {
    int numberOfReaders = 0;
    boolean busy = FALSE;
    condition okToRead, okToWrite;

    public:
    void startRead() {
        if (busy) okToRead.wait();
        numberOfReaders = numberOfReaders + 1;
        okToRead.signal();
    }
    void finishRead() {
        numberOfReaders = numberOfReaders - 1;
        if (numberOfReaders == 0) okToWrite.signal();
    }
    void startWrite() {
        if (numberOfReaders > 0 || busy) okToWrite.wait();
        busy = TRUE;
    }
    void finishWrite() {
        busy = FALSE;
        if (okToRead.queue())
            okToRead.signal();
        else
            okToWrite.signal();
    }
}
monitor ReadersWriters {
    int numberOfReaders = 0;
    boolean busy = FALSE;
    condition okToRead, okToWrite;

    public:
    void startRead() {
        if (busy || okToWrite.queue()) okToRead.wait();
        numberOfReaders = numberOfReaders + 1;
        okToRead.signal();
    }
    void finishRead() {
        numberOfReaders = numberOfReaders - 1;
        if (numberOfReaders == 0) okToWrite.signal();
    }
    void startWrite() {
        if (numberOfReaders > 0 || busy) okToWrite.wait();
        busy = TRUE;
    }
    void finishWrite() {
        busy = FALSE;
        if (okToWrite.queue())
            okToWrite.signal();
        else
            okToRead.signal();
    }
}
A Resource Allocation Monitor

```cpp
monitor ResourceAllocator {
    boolean busy = FALSE;
    condition resource;

    public:
    acquire(time: integer) {
        // time = How long is the resource needed for
        if (busy) resource.wait(time);
        busy = TRUE;
    }

    release() {
        busy = FALSE;
        resource.signal();
    }
}

-We assume priority scheduling for the wait queue.
```
Examples of using Monitors

Other interesting monitor examples:

▶ solve the producer consumer problem using monitors.
▶ an alarm clock monitor.
▶ a monitor that allows access to a file by no more than \( n \) processes.
▶ a monitor that allows access to a group of processes as long as the sum total of their process ids doesn’t exceed a certain integer, say \( n \).

The idea is to use the weakest scheduling that we can get away with.

(arbitrary \( < \) FIFO \( < \) Priority)
An Alarm Monitor

Write a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock, which invokes a procedure *tick* in your monitor at regular intervals.

```c
monitor AlarmClock {
    int now = 0;
    condition TimeToWakeup;
public:
    void SetAlarm (int period) {
        (* fill in the details here ... *)
    }
    void tick() {
        now++;
        TimeToWakeup.signal();
    }
}
```
An Alarm Monitor (solution)

```cpp
monitor AlarmClock{
    int now = 0;
    condition TimeToWakeup;
    public:
    void SetAlarm(int period){
        period = period + now; // at what time to wake up
        while (now < period) TimeToWakeup.wait(period);
        TimeToWakeup.signal(); // Wake up other processes due to wake up now
        // A cascaded wake up like the one we used in
        // the Readers and Writers problem
    }
    void tick(void) {
        now = now + 1;
        TimeToWakeup.signal();
    }
}
```

*Note: The variable now and period may overflow (if your operating system is not rebooted for many decades!). See synchronization-part2/monitors/java/alarm.monitor for a Java based solution.*
Writing Monitors with POSIX threads library

- The POSIX threads (Pthreads) library provides locks, semaphores and condition variables. Combining these concepts makes it possible to write monitors in C/C++.

- The idea is to have a monitor class (or class like file in C). For example, we may have `monitor.c` and `monitor.h`. Associate a mutex lock with it. In each method of the class, the first thing to do is to acquire the lock. We unlock the mutex before returning from each function.

- The condition variables in Pthreads library are the same as the condition variables used in the monitor concept.

- How to protect a class where public functions can call each other? Use a wrapper function for each public function. The wrapper function locks the mutex, calls the real function and then unlocks the mutex.

See example: `synchronization-part2/monitors/pthreads/account`
Monitors Support in POSIX threads

Mutexes and Semaphores have already been discussed previously. The following are the functions provided in Pthreads library for supporting condition variables.

```c
pthread_cond_t condvar;
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(...);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

See man pages for more information.

Notes on `pthread_cond_wait` and `pthread_cond_timedwait`: These functions always have an associated mutex (the one that is being used to enforce mutual exclusion in the monitor). These functions atomically release the mutex and cause the calling thread to block on the condition variable. Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.
/*synchronization-part2/monitors/pthreads/account/account.h*/
#ifndef __ACCOUNT_H
#define __ACCOUNT_H
#include <pthread.h>

typedef struct account Account;
typedef struct account * AccountPtr;

struct account {
    double balance;
    pthread_mutex_t mutex;
    void (*credit)(AccountPtr, double);
    void (*debit)(AccountPtr, double);
};

AccountPtr account_init();

#endif /* __ACCOUNT_H */
Account: A Monitor Example (contd)

/* synchronization-part2/monitors/pthreads/account/account.c */
/* appropriate header files */
#include <pthread.h>
#include "account.h"
static void credit(AccountPtr acct, double amount)
{
    pthread_mutex_lock(&acct->mutex);
    acct->balance += amount;
    pthread_mutex_unlock(&acct->mutex);
}
static void debit(AccountPtr acct, double amount)
{
    pthread_mutex_lock(&acct->mutex);
    acct->balance -= amount;
    pthread_mutex_unlock(&acct->mutex);
}
AccountPtr account_init()
{
    AccountPtr acct = (AccountPtr) malloc(sizeof(Account));
    acct->balance = 0.0;
    acct->credit = credit; acct->debit = debit;
    pthread_mutex_init(&acct->mutex, NULL);
    return acct;
}
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;
    }
    account = account_init();
    printf("initial balance = %lf\n", account->balance);
Account: A Monitor Example (contd)

/* synchronization-part2/monitors/pthreads/account/TestAccount.c */

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 = %1.1f\n", account->balance);
exit(0);
}

void *threadMain(void *arg)
{
    int i, amount;
    for (i=0; i<count; i++) {
        amount = 1;
        (*account->credit)(account, amount);
        /* account->credit(account, amount); okay in C99 */
    }
    pthread_exit(NULL);
}

Alarm Monitor in Pthreads

/* synchronization-part2/monitors/pthreads/alarm-clock/AlarmClock.h */
#ifndef __ALARMCLOCK_H
#define __ALARMCLOCK_H
#include <pthread.h>
typedef struct alarmclock AlarmClock;
typedef struct alarmclock * AlarmClockPtr;

struct alarmclock {
    long int now;
    pthread_mutex_t mutex;
    pthread_cond_t timeToWakeUp;
    void (*setAlarm) (AlarmClockPtr, long int);
    void (*tick) (AlarmClockPtr);
};

AlarmClockPtr alarmclock_init();

#endif /* __ALARMCLOCK_H */
Alarm Monitor in Pthreads

/* synchronization-part2/monitors/pthreads/alarm-clock/AlarmClock.c */
#include <stdlib.h>
#include "AlarmClock.h"
static void setAlarm(AlarmClockPtr, long int);
static void tick(AlarmClockPtr);

AlarmClockPtr alarmclock_init() {
    AlarmClockPtr alarm = (AlarmClockPtr) malloc(sizeof(AlarmClock));
    alarm->now = 0;
    alarm->setAlarm = setAlarm;
    alarm->tick = tick;
    pthread_mutex_init(&alarm->mutex), NULL);
    pthread_cond_init(&alarm->timeToWakeUp), NULL);
    return alarm;
}
void setAlarm(AlarmClockPtr alarm, long int period) {
    pthread_mutex_lock(&alarm->mutex));
    period = period + alarm->now;
    while (alarm->now < period)
        pthread_cond_wait(&alarm->timeToWakeUp), &alarm->mutex);
    pthread_cond_signal(&alarm->timeToWakeUp));
    pthread_mutex_unlock(&alarm->mutex));
}
void tick(AlarmClockPtr alarm) {
    pthread_mutex_lock(&alarm->mutex));
    alarm->now++;
    pthread_cond_signal(&alarm->timeToWakeUp));
    pthread_mutex_unlock(&alarm->mutex));
}
/* synchronization-part2/monitors/pthreads/alarm-clock/TestAlarm.c */
/* appropriate header files */
#include "AlarmClock.h"
#include "threadMain.h"

void *threadMain(void *);

pthread_t *tids;
AlarmClockPtr alarmClock;
int numThreads, count;

pthread_mutex_t mutex;
void signalHandler(int);"
...int get_my_index() {
    int i;
    pthread_mutex_lock(&mutex);
    for (i=0; i<numThreads; i++) {
        if (tids[i] == pthread_self()) {
            pthread_mutex_unlock(&mutex);
            return i;
        }
    }
    pthread_mutex_unlock(&mutex);
    return -1; /* we have a problem if we reach this statement */
}
void *threadMain(void *arg) {
    int i;
    int period;
    time_t current;
    int id = get_my_index();
    for (i=0; i<count; i++) {
        period = random() % 4 + 1; // 1 to 4 seconds
        time(&current);
        printf("Thread %d going to set up %d seconds alarm at %s", id, period, ctime(&current));
        (*alarmClock->setAlarm)(alarmClock, period);
        time(&current);
        printf("Thread %d waking up from %d seconds alarm at %s", id, period, ctime(&current));
    }
    pthread_exit(NULL);
}
void signalHandler(int signo) {
    alarmClock->tick(alarmClock);
    alarm(1); //reset alarm
}
Producers Consumers in Pthreads

/* Pthreads version of example originally from MSDN */
/* synchronization-part2/monitors/pthreads/producers-consumers/producers-consumers.c */
/* appropriate header files */
#define TRUE 1
#define FALSE 0
#define BUFFER_SIZE 10
#define PRODUCER_SLEEP_TIME_MS 500
#define CONSUMER_SLEEP_TIME_MS 2000
long Buffer[BUFFER_SIZE];
long LastItemProduced;
unsigned long QueueSize, QueueStartOffset;
unsigned long TotalItemsProduced, TotalItemsConsumed;
pthread_cond_t BufferNotEmpty, BufferNotFull, BufferLock;
int StopRequested;

int main (int argc, char* argv[])
{
    pthread_t hProducer1, hConsumer1, hConsumer2;
    pthread_cond_init (&BufferNotEmpty, NULL);
    pthread_cond_init (&BufferNotFull, NULL);
    pthread_mutex_init (&BufferLock, NULL);

    pthread_create (&hProducer1, NULL, ProducerThreadProc, (void *)1);
    pthread_create (&hConsumer1, NULL, ConsumerThreadProc, (void *)1);
    pthread_create (&hConsumer2, NULL, ConsumerThreadProc, (void *)2);
    puts("Press enter to stop...");
    getchar();
    pthread_mutex_lock (&BufferLock);
    StopRequested = TRUE;
    pthread_mutex_unlock (&BufferLock);

    pthread_cond_broadcast (&BufferNotFull);pthread_cond_broadcast (&BufferNotEmpty);
    pthread_join (hProducer1, NULL);pthread_join (hConsumer1, NULL);pthread_join (hConsumer2, NULL);

    printf("TotalItemsProduced: %lu, TotalItemsConsumed: %lu\n", TotalItemsProduced, TotalItemsConsumed);
    exit(0);
}
void *ProducerThreadProc (void * p)
{
    unsigned long ProducerId = (unsigned long)(unsigned long *)p;
    while (TRUE)
    {
        // Produce a new item (sleep in microseconds)
        usleep (rand() % PRODUCER_SLEEP_TIME_MS * 1000);

        pthread_mutex_lock (&BufferLock);
        unsigned long Item = LastItemProduced;
        while (QueueSize == BUFFER_SIZE && StopRequested == FALSE) {
            // Buffer is full - wait so consumers can get items.
            pthread_cond_wait (&BufferNotFull, &BufferLock);
        }
        if (StopRequested == TRUE) {
            pthread_mutex_unlock (&BufferLock);
            break;
        }
        // Insert the item at the end of the queue and increment size.
        Buffer[(QueueStartOffset + QueueSize) % BUFFER_SIZE] = Item;
        QueueSize++;
        TotalItemsProduced++;
        pthread_mutex_unlock (&BufferLock);

        // If a consumer is waiting, wake it.
        pthread_cond_signal (&BufferNotEmpty);
    }
    pthread_exit(NULL);
}
Producers Consumers in Pthreads (contd.)

```c
/* synchronization-part2/monitors/pthreads/producers-consumers/producers-consumers.c */

void *ConsumerThreadProc (void * p)
{
    unsigned long ConsumerId = (unsigned long)(unsigned long *)p;
    while (TRUE)
    {
        pthread_mutex_lock (&BufferLock);
        while (QueueSize == 0 && StopRequested == FALSE) {
            // Buffer is empty - sleep so producers can create items.
            pthread_cond_wait (&BufferNotEmpty, &BufferLock);
        }
        if (StopRequested == TRUE && QueueSize == 0) {
            pthread_mutex_unlock (&BufferLock);
            break;
        }
        // Consume the first available item.
        long Item = Buffer[QueueStartOffset];
        QueueSize--;
        QueueStartOffset++;
        TotalItemsConsumed++;
        if (QueueStartOffset == BUFFER_SIZE) {
            QueueStartOffset = 0;
        }
        pthread_mutex_unlock (&BufferLock);
        pthread_cond_signal (&BufferNotFull); // If a producer is waiting, wake it.
        // Simulate processing of the item. Argument in microseconds
        usleep (rand() % CONSUMER_SLEEP_TIME_MS * 1000);
    }
    pthread_exit(NULL);
}
```
How would you modify the producer consumer example so that it stops cleanly on receiving SIGTERM, SIGINT or SIGHUP signals?

SIGINT is generated by typing ctrl-c on the keyboard. SIGTERM or SIGHUP can generated sent via the `kill` command.

```
kill -s TERM pid
kill -s HUP pid
```

where the process can be found with the `ps augx` command.
Java threads are **preemptible**. Java threads may or may not be **time-sliced**. The programmer should not make any timing assumptions.

Threads have **priorities** that can be changed (increased or decreased).

This implies that multiple threads will have race conditions (read/write conflicts based on time of access) when they run. The programmer has to resolve these conflicts.

Example of a race condition: `Account.java`, `TestAccount.java` (In lab example folder: `synchronization-part2/monitor/java/account`)
Java Synchronization (2)

- Java has **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
// Only one thread can execute the update method at a time in the class.
synchronized void update() { //... }

// Access to individual datum can also be synchronized.
// The object buffer can be used in several classes, implying
// synchronization among methods from different classes.

synchronized(buffer) {
    this.value = buffer.getValue();
    this.count = buffer.length();
}
```

- Every Java object has an implicit monitor associated with it to implement the synchronized keyword. Inner class has a separate monitor than the containing outer class.

- Java allows **Reentrant Synchronization**, that is, a thread can reacquire a lock it already owns. For example, a synchronized method can call another synchronized method.
The `wait()` and `notify()` methods (of the `Object` class) allow a thread to give up its hold on a lock at an arbitrary point, and then wait for another thread to give it back before continuing. Another thread must call `notify()` for the waiting thread to wakeup. If there are other threads around, then there is no guarantee that the waiting thread gets the lock next. *Starvation* is a possibility. We can use an overloaded version of `wait()` that has a timeout. The method `notifyAll()` wakes up all waiting threads instead of just one waiting thread.
Example with `wait() / notify()`

```java
class MyThing {
    synchronized void waiterMethod() {
        // do something
        // now we need to wait for the notifier to do something
        // this gives up the lock and puts the calling thread to sleep
        wait();
        // continue where we left off
    }

    synchronized void notifierMethod() {
        // do something
        // notifier the waiter that we've done it
        notify();
        // do more things
    }

    synchronized void relatedMethod() {
        // do some related stuff
    }
}
```
Writing Monitors with Java

Java directly implements the notion of monitors with the synchronized keyword. Any Java class can be made into a monitor by following two rules.

- Add the `synchronized` keyword in front of each method declaration.
- Make sure that there are no directly accessible class variables. For example, make all class variables be private (which is the recommended Object-Oriented practice anyways).

Every java object has a `wait()`, `notify()`, `notifyAll()` methods that correspond to monitor concepts.
Synchronization Example 1

See examples in the folder:

synchronized-part2/monitors/java/account

- Race conditions: Account.java, TestAccount.java
- Thread safe version using synchronized keyword: RentrantAccount.java
A producer thread creates messages and places them into a queue, while a consumer thread reads them out and displays them.

The queue has a maximum depth.

The producer and consumer don’t operate at the same speed. In the example, the producer creates messages every second but the consumer reads and displays only every two seconds.

How long will it take for the queue to fill up? What will happen when it does?
Synchronization Example 2 (contd.)

See examples in the folder:
synchronized-part2/monitors/java/producers-consumers

- Producer and Consumer sharing a synchronized queue.
- See examples: SharedQueue\.java, Producer\.java, Consumer\.java, PC\.java
Synchronization Example 3: Alarm Monitor

- See example int the folder
  synchronized-part2/monitors/java/alarm.monitor
Condition variables support in MS Windows API


```c
CONDITION_VARIABLE condvar;
```

- **InitializeConditionVariable**: Initializes a condition variable.
- **SleepConditionVariableCS**: Sleeps on the specified condition variable and releases the specified critical section as an atomic operation.
- **SleepConditionVariableSRW**: Sleeps on the specified condition variable and releases the specified SRW lock as an atomic operation.
- **WakeAllConditionVariable**: Wakes all threads waiting on the specified condition variable.
- **WakeConditionVariable**: Wakes a single thread waiting on the specified condition variable.

CS stands for critical section. SRW stands for Slim Read Write locks.

*From the MSDN Library documentation.*
CRITICAL_SECTION CritSection;
CONDITION_VARIABLE ConditionVar;
void PerformOperationOnSharedData()
{
    EnterCriticalSection(&CritSection);
    // Wait until the predicate is TRUE
    while( TestPredicate() == FALSE )
    {
        SleepConditionVariableCS(&ConditionVar, &CritSection, INFINITE);
    }
    // The data can be changed safely because we own the critical section
    ChangeSharedData();
    LeaveCriticalSection(&CritSection);
    // If necessary, signal the condition variable by calling
    // WakeConditionVariable or WakeAllConditionVariable
    // so other threads can wake
}
/* From MSDN Library documentation */
/* ms-windows/synchronization-part2/producers-consumers.c */
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#define BUFFER_SIZE 10
#define PRODUCER_SLEEP_TIME_MS 500
#define CONSUMER_SLEEP_TIME_MS 2000

LONG Buffer[BUFFER_SIZE];
LONG LastItemProduced;
ULONG QueueSize;
ULONG QueueStartOffset;
ULONG TotalItemsProduced;
ULONG TotalItemsConsumed;

CONDITION_VARIABLE BufferNotEmpty;
CONDITION_VARIABLE BufferNotFull;
CRITICAL_SECTION BufferLock;
BOOL StopRequested;
Producers Consumers in MS Windows API (contd.)

```c
void __cdecl wmain (int argc, const wchar_t* argv[]) {
    InitializeConditionVariable (&BufferNotEmpty);
    InitializeConditionVariable (&BufferNotFull);
    InitializeCriticalSection (&BufferLock);

    DWORD id;
    HANDLE hProducer1 = CreateThread (NULL, 0, ProducerThreadProc, (PVOID)1, 0, &id);
    HANDLE hConsumer1 = CreateThread (NULL, 0, ConsumerThreadProc, (PVOID)1, 0, &id);
    HANDLE hConsumer2 = CreateThread (NULL, 0, ConsumerThreadProc, (PVOID)2, 0, &id);

    puts ("Press enter to stop...");
    getchar();

    EnterCriticalSection (&BufferLock);
    StopRequested = TRUE;
    LeaveCriticalSection (&BufferLock);

    WakeAllConditionVariable (&BufferNotFull);
    WakeAllConditionVariable (&BufferNotEmpty);

    WaitForSingleObject (hProducer1, INFINITE);
    WaitForSingleObject (hConsumer1, INFINITE);
    WaitForSingleObject (hConsumer2, INFINITE);

    printf ("TotalItemsProduced: %u, TotalItemsConsumed: %u\r\n", TotalItemsProduced, TotalItemsConsumed);
}
```
/* From MSDN Library documentation */
/* ms-windows/synchronization-part2/producers-consumers.c */

DWORD WINAPI ProducerThreadProc (PVOID p)
{
    ULONG ProducerId = (ULONG)(ULONG_PTR)p;

    while (true)
    {
        // Produce a new item.
        Sleep (rand() % PRODUCER_SLEEP_TIME_MS);
        ULONG Item = InterlockedIncrement (&LastItemProduced);

        EnterCriticalSection (&BufferLock);
        while (QueueSize == BUFFER_SIZE && StopRequested == FALSE) {
            // Buffer is full - sleep so consumers can get items.
            SleepConditionVariableCS (&BufferNotFull, &BufferLock, INFINITE);
        }
        if (StopRequested == TRUE) {
            LeaveCriticalSection (&BufferLock);
            break;
        }
        // Insert the item at the end of the queue and increment size.
        Buffer[(QueueStartOffset + QueueSize) % BUFFER_SIZE] = Item;
        QueueSize++;
        TotalItemsProduced++;
        LeaveCriticalSection (&BufferLock);
        // If a consumer is waiting, wake it.
        WakeConditionVariable (&BufferNotEmpty);
    }
    return 0;
}
/ From MSDN Library documentation */ /* ms-windows/synchronization-part2/producers-consumers.c */

DWORD WINAPI ConsumerThreadProc (PVOID p)
{
    ULONG ConsumerId = (ULONG)(ULONG_PTR)p;
    while (true)
    {
        EnterCriticalSection (&BufferLock);
        while (QueueSize == 0 && StopRequested == FALSE) {
            // Buffer is empty - sleep so producers can create items.
            SleepConditionVariableCS (&BufferNotEmpty, &BufferLock, INFINITE);
        }
        if (StopRequested == TRUE && QueueSize == 0) {
            LeaveCriticalSection (&BufferLock);
            break;
        }
        // Consume the first available item.
        LONG Item = Buffer[QueueStartOffset];
        QueueSize--;
        QueueStartOffset++;
        TotalItemsConsumed++;
        if (QueueStartOffset == BUFFER_SIZE) {
            QueueStartOffset = 0;
        }
        LeaveCriticalSection (&BufferLock);
        WakeConditionVariable (&BufferNotFull); // If a producer is waiting, wake it.
        // Simulate processing of the item.
        Sleep (rand() % CONSUMER_SLEEP_TIME_MS);
    }
    return 0;
}
Semaphores and Signals are useful for synchronization but not for sending information among processes.

Monitors allow communication using shared memory.

Message passing allows communication without using shared memory. The message passing mechanism copies the information from the address space of one process to the address space of another process.

What’s a message? How do we send a message? Why is the operating system needed as an intermediary to send messages? (for independent processes, since for threads we can always use shared memory).
Message Passing Using Mailboxes

- Mailboxes in user space versus in system space.
- *Message protocols.* A message may contain an instance of a C structure, may be a string of ASCII characters etc. The format is between the sender and receiver to decide a priori or negotiate.
- *Message headers* may contain information such as the sending process id, the receiver process id, number of bytes in the message, message type etc. Messages may not even have a header depending on how they are implemented.
- Performance issues. *Copy-on-write* mechanism can be used to improve the performance.
Message Passing Primitives

- **send().**
  - *synchronous send.* Blocks the sending process until the message is received by the destination process. This synchronization is an example of a producer-consumer problem.
  - *asynchronous send.* The message is delivered to the receiver’s mailbox and then the sender is allowed to continue without waiting for the receiver to read the message. The receiver doesn’t even have to ever retrieve the message.

- **receive().**
  - *blocking receive.* If there is no message in the mailbox, the process suspends until a message is placed in the mailbox. A receive operation is analogous to a resource request.
  - *non-blocking receive.* The process is allowed to query the mailbox and then control is returned immediately either with a message if there is one in the mailbox or with a status that no message is available.
Message passing and Semaphores

- How to simulate a semaphore using message passing?
- How to simulate a monitor using message passing?
- How to implement message passing using mailboxes (with all processes having access to shared memory) and semaphores?
Implementing Message Passing using Semaphores

- A shared mailbox area in shared memory will be used to hold the mailboxes. Each mailbox data structure contains an array of message slots. Each mailbox has a variable keeping track of the number of full slots and also has two queues that keep track of waiting processes/threads.
- Each process has an associated semaphore (initially zero) on which it will block when a send/receive must block.
- A global mutex is used to ensure mutual exclusion in accessing the shared mailbox area in memory.

Based on the above, now sketch out the pseudo-code for send/receive. How can we improve the performance of the above implementation?
Implementing Message Passing using Monitors

- Each mailbox is managed by a monitor. Each mailbox contains an array of message slots. Each mailbox has a variable keeping track of the number of full slots and also has two queues that keep track of waiting processes/threads.
- Each process has an condition variable on which it will block when a send/receive must block.

Based on the above, now sketch out the pseudo-code for send/receive.
Signals

Linux/Unix **signals** are a type of event. Signals are asynchronous in nature and are used to inform processes of certain events happening.

Examples:

- User pressing the interrupt key (usually **Ctrl-c** or **Delete** key). Generates the **SIGINT** signal.
- User pressing the stop key (usually **Ctrl-z**). Generates the **SIGTSTP** signal.
- The signal **SIGCONT** can restart a process if it is stopped.
- Signals for alarm, for hardware exceptions, for when child processes terminate or stop.
- Special signals for killing (**SIGKILL**) or stopping (**SIGSTOP**) a process. These cannot be ignored by a process.
For each signal there are three possible actions: default, ignore, or catch. The system call signal()/sigaction() attempts to set what happens when a signal is received. The prototype for the system call is:

```c
void (*signal(int signum, void (*handler)(int)))(int);
```

The above prototype can be made easier to read with a typedef as shown below.

```c
typedef void sighandler_t(int);
sighandler_t *signal(int, sighandler_t *);
```

The header file `<signal.h>` defines two special dummy functions `SIG_DFL` and `SIG_IGN` for use as signal catching actions. For example: `signal(SIGALRM, SIG_IGN);`
The system call `kill()` is used to send a specified signal to a specified process. For example:

```c
kill(getpid(), SIGSTOP);
kill(getpid(), SIGKILL);
```

Unreliable versus reliable signals. Blocking signals.

In Linux, when a signal is caught the signal handling is set to the default state. Thus we have to reset the signal handling as part of catching the signal. More control can be exercised by using the `sigaction()` and other related system calls.

Two main system calls deal with signals: `signal()` and `sigaction()`. The `sigaction()` call is newer and consistent across various systems whereas `signal()` is not necessarily consistent across systems but is simpler to use. See man page for details on both.
/* synchronization-part2/signal-ex1.c */
#include <signal.h>
static void onintr();

int main(void)
{
    if (signal(SIGINT, onintr) == SIG_ERR)
        printf("Unable to create signal handler\n");

    /* processing section */
    sleep(4);

    exit(0);
}

void onintr() // clean up on interrupt
{
    printf("Caught Ctl-C\n");
    exit(1);
}
Another Example of Using Signals

/* synchronization-part2/signal-ex2.c */
#include <signal.h>
#include <stdio.h>
static void sig_handler(int);

int main(void) {
    int i, status;
    pid_t parent_pid, child_pid;

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("Parent: Unable to create handler for SIGUSR1\n");

    if (signal(SIGUSR2, sig_handler) == SIG_ERR)
        printf("Parent: Unable to create handler for SIGUSR2\n");

    parent_pid = getpid();
    if ((child_pid = fork()) == 0) {
        kill(parent_pid, SIGUSR1);
        for (; ;) pause();
    } else {
        kill(child_pid, SIGUSR2);
        sleep(2);
        printf("Parent: Terminating child...\n");
        kill(child_pid, SIGTERM);
        wait(&status);
        printf("done\n");
    }
}
static void sig_handler(int signo)
{
    switch(signo) {
    case SIGUSR1:
        printf("Parent: received SIGUSR1\n");
        fflush(stdout);
        break;
    case SIGUSR2:
        printf("Child: received SIGUSR2\n");
        fflush(stdout);
        break;
    default: break;
    }
    return;
}
Signal Handling after Exec

When a program is exec’d the status of all signals is either default or ignore. The exec functions changes the disposition of any signals that are being caught to their default action (why?) and leaves the status of all other signals alone. For example:

```c
if (fork()==0) { // the child process
    if (signal(SIGINT, SIG_IGN) == SIG_ERR)
        err_ret("failed to set SIGINT behavior");
    if (signal(SIGTSTP, SIG_IGN) == SIG_ERR)
        err_ret("failed to set SIGTSTP behavior");
    execvp(program, argv);
    // the exec'd program will ignore the
    // signals SIGINT and SIGTSTP
```
How an application ought to set its signal handling:
An application process should catch the signal only if the signal is not currently being ignored.

```c
int sig_int(), sig_quit();

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```
/* synchronization-part2/signla-ex3.c */
#include <signal.h>
#include <setjmp.h>
jmp_buf env;
static void onintr();
int main(void)
{
    int code;
    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        if (signal(SIGINT, onintr) == SIG_ERR)
            printf("Error in setting up interrupt handler\n");
    code = sigsetjmp(env, 1);
    printf("before the loop\n");
    for (;;) {
        /* main processing loop */
    }
}
#define FROM_ONINTR 333333
void onintr() {
    // re-establish signal handler
    if (signal(SIGINT, onintr) == SIG_ERR)
        printf("Error in setting up interrupt handler\n");
    printf("Caught Interrupt\n");
    // long jump back to before the main loop.
    siglongjmp(env,FROM_ONINTR);
}
Other Signal Issues

- Signals are set to their default after being caught (under Linux and System V UNIX semantics). This can be changed by using `sigaction()` instead of the `signal()` system call.

- What if the program was updating some complicated data structures... then `exit` or `siglongjmp()` is not a good solution. We can set a flag to indicate the interrupt and continue processing. The program can deal with the interrupt later.

- Under the POSIX semantics a process can also block signals so that they are delivered later when it is ready to deal with the signals. (See man pages for `sigaction()` and `sigprocmask()` under Linux).

- What if the parent and the child are both trying to catch an interrupt? They both might then try to read from standard input. That would be confusing, to say the least. The solution is to have the parent program ignore interrupts until the child is done.
/ * signal-ex4.c */ / * appropriate header files */ void signalHandler(int sig); jmp_buf env; int count=0; int main(int argc, char **argv) { int code; unsigned int *ptr;  

if (signal(SIGSEGV, signalHandler) == SIG_ERR) 
   printf("Could not setup the signal handler\n"); count = atoi(argv[1]); printf("count = %d\n",count); 

// hey try another stab at places where we shouldn't be!
// this might cause a segmentation violation!!!
code = sigsetjmp(env,1); if (code == 1) { // returning from the signal handler, change the pointer 
   ptr=(int *) random(); printf("trying memory location [%x]\n",(unsigned int) ptr); 
   fflush(NULL); 
} 
   *ptr = 10; exit(0); 
}     
void signalHandler(int sig) { 
   count--; if (count < 0) exit(1); sleep(1); // just to make it easier to watch it happen 
   printf("Caught segmentation violation\n"); 
   fflush(NULL); siglongjmp(env,1); }
/ * synchronization-part2/signal-ex5.c */
/* appropriate header files */
static void onintr();

int main(void)
{
    sigset_t newmask, oldmask, pendmask;
signal(SIGINT, onintr);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("Error in blocking signals\n");

/* processing section: SIGINT remains pending */
sleep(5);

if (sigpending(&pendmask) < 0)
    err_sys("sigpending error");
if (sigismember(&pendmask, SIGINT))
    printf("\n Signal SIGINT is pending\n");
if (sigprocmask(SIG_UNBLOCK, &newmask, &oldmask) < 0)
    err_sys("Error in unblocking signals\n");

/* better to restore the old mask, since the signal might be already blocked blocked before here (if this code was part of a function that can be called from various places)
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) <0)
        err Sys("Error in unblocking signals\n"); */
exit(0);
}
void onintr()
{
    printf("Caught Ctl-C\n");
exit(1);
}
Signal Handling in the Shell

» How to handle Ctrl-c or SIGINT?
  » The shell ignores it on startup.
  » The shell catches it on startup and sends appropriate signal to foreground jobs when it catches the signal.
  » The shell catches it on startup, then ignores it when starting a foreground job and catches it again afterwards.

» How to handle Ctrl-z or SIGTSTP?
  » The shell ignores it on startup.
  » The shell catches it on startup and send appropriate signal to foreground jobs when it catches the signal.
  » The shell catches it on startup, then ignores it when starting a foreground job and catches it again afterwards.

» How do we prevent background jobs from getting affected by Ctrl-c and Ctrl-z?

» How to implement fg and bg built-in commands?
Pipes in Unix

- A pipe allows communication between two processes that have a common ancestor.
- A pipe is a half-duplex (data flows in only one direction) FIFO buffer with an API similar to file I/O.
  ```c
  #include <unistd.h>
  int pipe(int filedes[2]);
  ```
- Reading from a pipe whose write end has been closed causes an End Of File to be returned. Writing to a pipe whose read end has been closed causes the signal SIGPIPE to be generated. The write returns with errno set to EPIPE.
- The size of pipe is limited to PIPE_BUF. A write of PIPE_BUF or less will not interleave with the writes from other processes. The constant PIPE_BUF is defined in the file /usr/include/linux/limits.h
The Power Of Pipelines

Find the 10 most frequent words in a given text file (and their respective counts).

cat Shakespeare.txt | tr -cs "[A-Z][a-z][']" "[\012*]" | tr A-Z a-z | sort | uniq -c | sort -rn | sed 10q
Generate all anagrams from a given dictionary.

```
sign < $2 | sort | squash | awk '{if (NF > 1) print $0}"

---squash---
#!/bin/sh
/usr/bin/awk '
$2 != prev { prev = $2; if (NR > 1) printf "\n"}
    { printf " %s ", $2 }
END    { printf "\n" }'

---sign.c---
#include <stdio.h>
#include <string.h>
#define WORDMAX 101
int compchar(char *x, char *y) { return ((*x) - (*y));}
void main(void)
{
    char thisword[WORDMAX], sign[WORDMAX];

    while (scanf("%s", thisword) != EOF) {
        strcpy(sign, thisword);
        qsort(sign, strlen(sign), 1, compchar);
        printf("%s %s\n", sign, thisword);
    }
}
A Pipe Example

#include "ourhdr.h"
int main(void)
{
int n, fd[2];
pid_t pid;
char line[MAXLINE];

if (pipe(fd) < 0)
   err_sys("pipe error");
if ((pid = fork()) < 0)
   err_sys("fork error");
else if (pid > 0) { /* parent */
close(fd[0]);
write(fd[1], "blup bup bubble bubble world\n", 12);
} else { /* child */
close(fd[1]);
n = read(fd[0], line, MAXLINE);
// STDOUT_FILENO is defined in /usr/include/unistd.h
write(STDOUT_FILENO, line, n);
}
exit(0);
```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void main()
{
    int fd;
    if ((fd = open("log", O_WRONLY|O_CREAT, S_IRWXU)) == -1)
        printf("open failed\n");

    close(1); // close stdout
    dup(fd); // make stdout point to the file fd

    // now this print goes to the file log
    printf("I am redirected\n");
}
```
newfd = dup(1);
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    int pipefd[2];

    /* Assumes file descriptor 0 and 1 are open */
    pipe (pipefd);
    if ((pid = fork()) == (pid_t)0) /* child process */
    {
        close(1); /* close stdout, normally fileno 1 */
        dup(pipefd[1]); /* points pipefd at file descriptor */
        close (pipefd[0]);
        execlp ("ls", "ls", (char *)0); /* child process does ls */
    } else if (pid > (pid_t)0) {
        close(0); /* close stdin, normally fileno 0 */
        dup (pipefd[0]);
        /* point the child's stdout to parent's stdin */
        close (pipefd[1]);
        execlp ("sort", "sort", (char *)0); /* parent process does sort */
    }
    exit(0);
}
/* appropriate header files */

int main()
{
    pid_t pid1, pid2;
    int pipefd[2];
    int status;

    if ((pid1 = fork()) == (pid_t)0) /* first child process */
    { // Assumes file descriptors 0 and 1 are open
        pipe (pipefd);
        if ((pid2 = fork()) == (pid_t) 0) { // second child process
            close(1); /* close stdout, normally fileno 1 */
            dup(pipefd[1]); /* points pipefd at file descriptor */
            close (pipefd[0]);
            execlp("ls", "ls", (char *)0); /* child process does ls */
        } else {
            close(0); /* close stdin, normally fileno 0 */
            dup(pipefd[0]);
            close (pipefd[1]);
            execlp("sort", "sort", (char *)0); /* child does sort */
        }
    }
    // parent waits for the last process in the pipeline to finish
    waitpid(pid1, &status, 0);
    exit(0);
}
// include appropriate header files
void main()
{
    pid_t pid1, pid2, pid3;
    int pipefd23[2], pipefd12[2];
    int status;
    
    if ((pid1 = fork()) == 0) /* first child process */
    {
        pipe (pipefd23);
        if ((pid2 = fork()) == 0) { // second child process
            pipe (pipefd12);
            if ((pid3 = fork()) == 0) { // third child process
                close(1); /* close stdout */
                dup(pipefd12[1]); /* points pipefd at file descriptor */
                close (pipefd12[0]);
                execlp ("ls", "ls", (char *)0); /* child process does ls */
            } else {
                // this is the middle process in the pipeline
                close(1); /* close stdout */
                dup(pipefd23[1]); /* points pipefd at file descriptor */
                close (pipefd23[0]);
                
                close(0); /* close stdin, normally fileno 0 */
                dup(pipefd12[0]);
                close (pipefd12[1]);
                execlp ("sort", "sort", (char *)0); /* child process does ls */
            }
        } else {
            close(0); /* close stdin, normally fileno 0 */
            dup(pipefd23[0]);
            close (pipefd23[1]);
            execlp ("wc", "wc", (char *)0); /* child does sort */
        }
    } else {
        close(0); /* close stdin, normally fileno 0 */
        dup(pipefd23[0]);
        close (pipefd23[1]);
        execlp ("wc", "wc", (char *)0); /* child does sort */
    }
}

waitpid(pid1, &status, 0);
A process group is a collection of one or more processes. Each process group has a unique process group ID.

Each process under Unix belongs to a process group. Each process group may have a process group leader. The leader is identified by having its process group ID equal to its process ID.

The two related systems calls allow a process to find out its process group’s ID and to change its process group.

```c
#include <sys/types.h>
#include <unistd.h>

pid_t getpgrp(void);
int set setpgid(pid_t pid, pid_t pgid);
```

A process can set the process group ID of itself or its children. Furthermore it cannot change the process group ID of its child after the child calls the `exec()` system call.

Only processes in the foreground group can perform I/O from the terminal. A process that is not in the foreground group will receive a `SIGTTIN` or `SIGTTOU` signal when it tries to perform I/O on the terminal, which will stop the process.

See several examples in the folder `process-groups` under `synchronization-part2` lab folder.
Named Pipes (FIFOs)

- Named Pipes (FIFOs) allow arbitrary processes to communicate.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo ( const char *pathname, mode_t mode );
```

```
mkfifo pathname
```

- If we write to a FIFO that no process has open for reading, the signal `SIGPIPE` is generated. When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.
- The reads/writes can be made blocking or non-blocking.
- If we have multiple writers for a FIFO, atomicity is guaranteed only for writes of size no more than `PIPE_BUF`. 
Uses of FIFOs

- Can be used by shell commands to pass data from one shell pipeline to another, without creating intermediate temporary files.

```bash
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

A real example of a nonlinear pipeline:

```bash
wc < fifo1 &
cat /usr/share/dict/words | tee fifo1 | wc -l
```

- Look at the following simple example using one fifo:

```bash
lab/pipes-fifos/hello-fifo.c
```

- Look at the following example for a two-way communication using two fifos: `lab/pipes-fifos/fifo-talk.c`
The server creates a FIFO using a pathname known to the clients. Clients write requests into this FIFO.

The requests must be *atomic* and of size less than `PIPE_BUF`, which is defined in `limits.h` standard header file.

The server replies by writing to a client-specific FIFO. For example, the client specific FIFO could be `/tmp/serv1.xxxxx` where `xxxxx` is the process id of the client.

See [wikipedia](https://en.wikipedia.org) for more information on the client-server model.
Fifo Server Client Example

- Shows a simple server that listens on a fifo to any client that connects to it and chats with the client.
- Server code: lab/pipes-fifos/fifo-server.c
- Client code: lab/pipes-fifos/fifo-client.c
Daemons and Servers

➢ A **Daemon** is a process that lives for a long time. Daemons are often started when the system is bootstrapped and terminate only when the system is shutdown. They run in the background because they don’t have a controlling terminal.

➢ A **Server** is a process that waits for a client to contact it, requesting some type of service. Typically, the server then sends some reply back to the client.

➢ A common use for a daemon process is as a server process.
Writing a Daemon

Coding rules for a daemon.

1. Call \texttt{fork()} and have the parent exit.

2. Call \texttt{setsid} to create a new session. Then the process becomes a session leader of a new session, becomes the process group leader of a new process group and has no controlling terminal.

3. Change the current working directory to the root directory. Alternately, some daemons might change the working directory to some specific location.

4. Set the file creation mask to 0, so it does not use anything inherited from its parent process.

5. Unneeded file descriptors should be closed.

6. Use the \texttt{syslog} system call to log messages to the \texttt{syslogd} daemon, which is a central facility for all daemons to log messages. See man page for syslog (\texttt{man 3 syslog}) for more details.
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int daemon_init(void)
{
    pid_t pid;

    if ( (pid = fork()) < 0)
        return(-1);
    else if (pid != 0)
        exit(0); /* parent goes bye-bye */

    /* child continues */
    setsid();   /* become session leader */

    chdir("/");   /* change working directory */

    umask(0);   /* clear our file mode creation mask */

    return(0);
}
Synchronization in the Linux kernel

- **Atomic operations** on bits, 32-bit and 64-bit integers. See `include/asm-generic/atomic.h` and `arch/x86/include/asm/atomic.h` for architecture specific implementation. See `arch/x86/include/asm/bitops/atomic.h` for test-and-set related bit operations.
- **Spinlocks.** Threads do not sleep if waiting for a lock. Suitable only for short duration locks.
- **Reader-Writer spinlocks.** Gives preference to readers over writers. Multiple readers can hold the lock but only one writer.
- **Semaphores.** See `include/linux/semaphore.h` and `kernel/locking/semaphore.c`. Uses wait queues and sleep.
- **Reader-Writer Semaphores.**
- **Mutexes.** Similar to a binary semaphore but with a simpler interface, more efficient performance, and additional constraints on its use.
- **Completion variables.** Similar to condition variables.
- **Sequential locks.** Reader and writers with preference given to writers.
- **Read-Write barriers and ordering.**
- **BKL: Big Kernel Lock.** Removed in kernel version 2.6.39.
- **Preemption disabling.** Bad thing :-(
