CSCI 3411 – OS Lab 4

SYNCHRONIZATION

Overview

- Review – Function Pointers and Pthreads
- Mutex
- Semaphore
- Monitor

C Function Pointers

- Declaration: `returnType (* ptrName) (paramList);`
  - Ex. `int (* myFuncPtr)(int, int);`
    - Declares a function pointer which can only point to a function that takes two int parameters and returns at int.
- Assignment: `myFuncPtr = &myFunction;`
- Dereference: `(* myFunPtr)(5, 11);`

C Function Pointers

- Why can't you declare a function pointer like these:
  - `int *myFuncPtr(int, int);`
  - `int *(myFunPtr(int, int));`
- Why can't you deference the function pointer:
  - `*myFuncPtr(0, 10);`
- Answer: C's Operator Precedence
  - Function Calls (i.e. (…)) are evaluated before * operator.
  - * operator binds to the type by default, not the variable name.

Pthreads

- Threading mechanism for C (POSIX threads).

- Usage:
  1. Create variables to hold thread ID numbers.
  2. Create thread attributes (can leave at defaults – priority level, etc.)
  3. Create and start threads:
     - Supply what function each thread should run and what arguments to pass to these functions.
  4. (Optionally) let the parent wait until all threads are finished.

Pthreads

- Usage:
  1. Include `<pthread.h>`
  2. Create `pthread_t` variables to hold ID numbers (one for each thread).
  3. Create `pthread_attr_t` variable and set it to default values.
     - `pthread_attr_init(pthread_attr_t *)`
  4. Create and start threads:
     - `pthread_create (pthread_t *, pthread_attr_t *, void *(* name)(void *), void *)`
     - In order: variable for ID number, attributes, function to execute, arguments for the function.
  5. (Optional) let the parent wait until all threads are finished.
     - `pthread_join(pthread_t, NULL);`

Mutex

- Mutual Exclusion
- Locking mechanism – `acquire()` and `release()`
  - Built-in "boolean" variable.
- "Waiting" is implemented via a busy loop / spinlock.
  - Effective if the time to context switch > busy waiting.

Mutex

- Usage:
  1. Call `acquire()` to obtain the mutex lock.
     - If someone else has it, wait.
  2. Execute code in critical section.
  3. Call `release()`

```
mutex.acquire();
/* Critical section code */
mutex.release();
```

Mutex in C

- Included in `<pthread.h>`
- `pthread_mutex_t` - defines a mutex type

- `pthread_mutex_init (pthread_mutex_t *, phtread_mutex_attr *)`
  - Call to initialize a mutex. The second param can be NULL => defaults.

- `pthread_mutex_lock (pthread_mutex_t *)`
  - Same as `acquire()`

- `pthread_mutex_unlock (pthread_mutex_t *)`
  - Same as `release()`

- `pthread_mutex_destroy (pthread_mutex_t)`

Mutex in C - Usage

```c
#include <pthread.h>

pthread_mutex_t mutex;
int main(){
    pthread_mutex_init(&mutex);
    …
    // Create, run, and join threads
    …
    pthread_mutex_destroy(&mutex);
}
```

```c
void *threadFunc(void *params){
    pthread_mutex_lock(&mutex);

    // Critical section

    pthread_mutex_unlock(&mutex);
}
```

Semaphores

- Can be used to control resource access for N resources.
  - Example: a system to reserve study rooms in the library.
- `wait()` and `signal()`.

- Study room semaphore initialized to 10
  - Calling wait decrements the value by 1, unless there are none left, in which case you wait for one to become available.
  - Calling signal increases the value by 1.
  - As opposed to having one mutex per room.

- A semaphore initialized to 1 == mutex.

Semaphores

- Usage:
  1. Initialize a semaphore to N (to represent N resources or N allowable accesses).
  2. Call `wait()` to decrement a semaphore's value by one.
     - If <= 0, wait until the resource is available.
  3. Execute critical section.
  4. Call `signal()` to increment a semaphore's value by one.

Semaphore Implementation

- Wait can be implemented with a waiting loop, like the mutex.
- Alternative to waiting loop:
  - Each semaphore keeps track of a process queue.
  - In `wait()` – instead of causing a process to spinloop, add it to the process queue and suspend its execution.
  - In `signal()` – wakeup the process at the head of the queue.

Semaphores in C

- Included in `<semaphore.h>`
- `sem_t` – defines a semaphore type
- `sem_init (sem_t *, int, int)`
  - The first int is a flag to set if it should be shared with a forked process.
  - The second is the semaphore's initial value.
- `sem_wait (sem_t *)`
- `sem_post (sem_t *)`
  - Same as `signal()`
- `sem_destroy (sem_t *)`
- `sem_value (sem_t *, int *)`
  - Retrieves the current value of the semaphore.

Semaphores in C - Usage

```
#include <semaphore.h>

sem_t semaphore;
int main(){
    sem_init(&semaphore, 0, 5);
    …
    // Create, run, and join threads
    …
    sem_destroy(&semaphore);
}
```

```
void *threadFunc(void *params){
    sem_wait(&semaphore);

    // Critical section

    sem_signal(&semaphore);
}
```

Monitors

- Encapsulates programmer-defined functions and variables into one synchronized data type (a module, class, object, etc.).
- Special "condition" variables that can be used with wait() or signal()

- Only one process is allowed at a time within a monitor.
- Multiple processes can be waiting in a queue to enter or waiting on specific condition variables.

Monitors

```
monitor myMonitor {
    int localVar1;
    char localArr1[];
    condition x, y, z;

    void fun1() {
        …
        x.wait();
        y.signal();
    }

    void fun2 () {
        y.wait();
        …
    }

    void fun3 () {
        x.signal();
        …
    }
}
```

<u>Usage</u>

```
Thread 1:
    myMonitor.fun1(); // Waiting on x
    printf(“A\n”);
Thread 2:
    myMonitor.fun2(); // Waiting on y
    printf(“B\n”);
Thread 3:
    myMonitor.fun3(); // Signals X
    printf(“C\n”);
```

Monitors

- Differences from other synchronization constructs:
  - A monitor includes synchronized functions and variables.
  - Only one process allowed inside at a time (unlike a semaphore).
    - If nobody is waiting on a condition, calling its `signal()` doesn't do anything.
  - Less room for erroneously calling wait / signal.
    - With semaphores:
      - Calling `signal(…)` before `wait(…)`
      - Calling `wait(…)` and then `wait(…)` again instead of `signal(…)`
      - Omitting `wait(…)` or `signal(…)`

Monitors

- Process P calls `signal()` is called on a condition variable which Process Q is waiting on. When do you let Q inside the monitor?
  - Signal and wait: Q is let inside immediately and P waits until Q leaves (or waits).
  - Signal and continue: Q waits until P leaves (or waits).

Monitor Implementation

- Monitors are really only a concept (an abstract data type). At the end of the day, you need some way to provide the synchronization features….
  - Implementation via semaphores (pg. 229):
    1. Semaphore to guard entrance to each function.
    2. Semaphore(s) to guard access to each condition variable.
    3. Semaphore used to implement "signal and wait".

Questions?