Seminar 5: Assignment 3

Reminder

- Assignment 3 Due: 2017/03/27 @ 23:55

- Assignments must be written in C

- This assignment does **NOT** require actually assigning memory

Part 1 - Dining Philosophers

- HINT: Use threads, it's much easier than using processes

- <pthread.h> is the library

- Also need to add a compiler flag, -pthread

- <pthread.h> library also includes the functions needed for mutexes

Threads

- pthreads lets you call a function in your program that will run concurrently
- Any global variables in your program will be shared between threads
- Lots of functions, but you will mostly need:
  - pthread_create - self explanatory
  - pthread_exit - call from a thread to exit
  - pthread_join - like waitpid but for threads

Shared Memory & Mutexes

- So, threads can share global memory
- But you would not want them to try to write to the same location at the same time or in an unpredictable order.
- A mutex variable acts like a "lock" protecting access to a shared data resource like a global variable.
- Only one thread can lock (or own) a mutex variable at any given time - allows for ordering of access to the variable.
- If a thread owning a mutex is updating a global variable it can ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the updating. The variable being updated belongs to a "critical section".

Using a Mutex

- Create and initialize a mutex variable.
- Several threads attempt to lock the mutex.
- Only one succeeds and that thread owns the mutex.
- The owner thread performs some set of actions.
- The owner unlocks the mutex.
- Another thread acquires the mutex and repeats the process.
- Finally the mutex is destroyed.

Mutex Variables

- Mutex variables must be declared with type pthread_mutex_t, and must be initialized before they can be used.
- pthread_mutex_init
- pthread_mutex_destroy
- pthread_mutexattr_init
- pthread_mutexattr_destroy

Locking and Unlocking

- Functions you'll need for this are:
  - pthread_mutex_lock
  - pthread_mutex_trylock
  - pthread_mutex_unlock

Part 2 - Memory Management Simulator

- You'll be writing a memory management simulator
- You'll have 128MB of 'memory', and each process will take up a certain number of MBs
- Processes will be simulated with structs
- Implementing 4 memory management algorithms: first-fit, next-fit, best-fit, worst-fit

Process Struct

- Process structs will be in a queue
- Each process will need to know:
  - It's ID and memory usage
  - # of times it has been swapped out of memory
- You'll need to keep track of how long each process has been in memory

First-Fit

- Probably the simplest algorithm, good place to start
- Simply check if there is any space for the process, and place it in that space
- If no space is found, remove oldest process and check again

First-Fit Pseudocode

```
while(insertingProcess) {
   for (i < memory) {
      if (memory[i] == 0) {
         space++;
         if (space >= processMemory) {
            insertProcess();
            break;
         }
      }
   }

   // No space found
   if (noSpace) {
      // remove oldest process
   }
}
```

Best-Fit

- Places a process in the smallest block of unallocated memory in which it will fit
- So we have to find all of the holes in memory, and choose the best one
- If we have a 12 MB process and:
    - 14 MB hole
    - 19 MB hole
    - 11 MB hole
    - 13 MB hole
- Which hole will the process go in to?

Best-Fit Pseudocode

```
while(insertingProcess) {
    for (i < memory) {
        if (memory[i] == 0) {
            space++;
        }
        else {
            if (space < smallestSpace && space >= processSize) {
                smallestSpace = space;
            }
        }
    }

    // No space found
    if (noSpace) {
        // remove oldest process
    }
    else {
        insertProcess()
    }
}
```

Worst-Fit

- Places a process in the largest block of unallocated memory in which it will fit
- So (again) we have to find all of the holes in memory, and choose the best one
- If we have a 12 MB process and:
    - 14 MB hole
    - 19 MB hole
    - 11 MB hole
    - 13 MB hole
- This time it will go into the 19 MB hole

Worst-Fit Pseudocode

```
while(insertingProcess) {
    for (i < memory) {
        if (memory[i] == 0) {
            space++;
        }
        else {
            if (space > largestSpace && space >= processSize) {
                largestSpace = space;
            }
        }
    }

    // No space found
    if (noSpace) {
        // remove oldest process
    }
    else {
        insertProcess()
    }
}
```

Next-Fit

- A variant of first fit
- We keep track of the last place we inserted memory, and insert the next process into the next hole
- Attempts to stop small holes accumulating at the start of memory

Next-Fit Pseudocode

```
while(insertingProcess) {
    for (i = lastIndex < memory) {
        if (memory[i] == 0) {
            space++;
            if (space >= processMemory) {
                insertProcess();
                lastIndex = i;
                break;
            }
        }
        if (i >= memory) {
            i = 0;
        }
    }

    // No space found
    if (noSpace) {
        // remove oldest process
    }
}
```