Discussion 5
Memory

Yibo Wang
June 1, 2007

Homework 2- 6.30(1)
class DiningServer{
    enum State{THINKING, HUNGRY, EATING};
    State[] states = new State[5];
    Condition[] self = new Condition[5];
    ReentranLock lock;
    public DiningServer{
        lock = new ReentranLock();
        for (int i=0; i<5; ++i){
            states[i] = State.THINKING;
            self[i] = lock.newCondition();
        }
    }
    ...
}

Homework 2- 6.30(2)
public void takeChopsticks(int id){
    int i = id;
    lock.lock();
    states[i] = State.HUNGRY;
    try{
        test(i);
        if (states[i] != State.EATING){
            self[i].await();
        }
    } finally{
        lock.unlock();
    }
}

Homework 2-6.30(3)
class Philosopher implements Runnable{
    ... // other variables
    private DiningServer dining_server;
    ... // initialize
    public void run(){
        ...
        dining_server.takeChopsticks(pid);
        ...
        dining_server.putChopsticks(pid);
        ...
    }
    ...
}

A left question
• Are semaphores and monitors equivalent insofar as they can be used to implement the same types of synchronization?
  – could one be implemented by the other?
• Yes, because...

Reason for equivalent
• A semaphore can be implemented by a monitor
  monitor semaphore {
      int value = 0;
      condition c;
      semaphore increment() {
          value++;
          c.signal();
      }
      semaphore decrement() {
          while (value == 0) c.wait();
          value--;
      }
  }
• A monitor can be implemented by a semaphore
  in this way:
  – Each condition variable is represented by a blocking semaphore
  – Each thread has a semaphore associated with its queue entry.
  – When a thread performs a wait operation, it creates a new semaphore (initialized to zero)
  – Appends the semaphore to the queue associated with the condition variable
  – Performs a blocking semaphore decrement on the newly created semaphore.
  – When a thread performs a signal on a condition variable, the first process in the queue is awakened by performing an increment on the corresponding semaphore.
Memory allocation algorithm

- Main allocation algorithm
  - first-fit
  - best-fit
  - worst-fit
- Given several partitions and several processes, you should be able to allocate spaces to each process with different algorithm

A Paging problem

- Given a virtual memory size of $2^{30}$ words, a physical memory size of $2^{20}$ words, and a page size of $4K = 2^{12}$ words:
  - a. How many pages in virtual memory?
    - $2^{30}/2^{12} = 2^{18}$
  - b. How many frames in physical memory?
    - $2^{20}/2^{12} = 2^{8}$
  - c. If my process has $2^{22}$ words and each page map table (PMT) entry takes 2 bytes, how big is my PMT?
    - $2^{22}/2^{12} = 2^{10}$ pages
    - $2^{10} \times 2 = 2^{11}$ bytes
  - d. What is in the PMT? (what are those 2 bytes?)
    - frame number
  - e. What is the input to access the PMT?
    - page number
  - f. What is the output from accessing the PMT?
    - frame number
  - g. How many bits in the displacement field of a virtual address? 12
  - h. How many bits in the displacement field of a physical address? 12

Main Memory

- Internal and external fragmentation
  - Internal fragmentation is the area in a page that is not used by the job occupying that region or page.
  - External fragmentation is the area in the whole memory space that is too small to be allocated to any job

Memory organization schemes

- Contiguous
  - External fragmentation
- Segmentation
  - External fragmentation
  - Code sharing
- Paging
  - Internal fragmentation
  - Code sharing
Virtual Memory

- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes
- Allows for more efficient process creation

Demand Paging

- Bring a page into memory only when it's needed

Page Fault

Page Replacement

- Want lowest page-fault rate
- Algorithm
  - OPT
  - FIFO
  - LRU
  - SECOND CHANCE
  - COUNTING ALGORITHMS
  - PRIORITY

Thrashing

- Thrashing is caused by under allocation of the minimum number of pages required by a process, forcing it to continuously page fault.
- The system can detect thrashing by evaluating the level of CPU utilization as compared to the level of multiprogramming.
- It can be eliminated by reducing the level of multiprogramming.

Adding System call (1)

- Write your function
- Define a system call number and add it to /usr/src/linux-2.x/include/asm-i386/unistd.h
  - System call number: index into a table of pointers pointed to the starting address of the system call
- The table of pointers stored in /usr/src/linux-2.x/arch/i386/kernel/entry.S
- Add your file to the corresponding Makefile
Adding System call (2)

Build the kernel

- Obtain the source code
- Compile the kernel
  - `make xconfig/oldconfig/menuconfig`
  - `make dep`
  - `make bzImage`
- Add a new entry to the bootable kernels
  - `lilo`
  - `grub`

Adding System call (3)

Using the newSystem Call

- Use `syscall0()` to invoke the newly defined system call
  - `_syscall0(int, helloworld);`
  - first argument: type of value returned by the system call
  - second argument: the name of the system call
- Then use it

Adding a system call (4)

- IP address of linux box we provided:
  - 128.223.6.137
- students should be able to 'ssh' into that machine
- account name is the email account name you used
- password is NULL.
- after login, please change your passwd using 'passwd' ASAP.
- Source code is located at:
  - `/home/linux-2.6.21.tar.gz`

Adding a system call (5)

What your should do

- Follow the bold step shown before
- leave all things you revised or created under your directory
- you still need to submit the required documents through
  - `turnin`
  - `email`
- You can do more and will receive extra points