CS330 Assignment 3

Aman Singh Y9066
Amitesh Maheshwari Y9078
Aniruddh Vyas Y9086
Gaurav Krishna Y9224

October 24, 2011

Objective

Part 1: Shared Memory

Implementation of the following system calls to set up shared memory:

- shared_memory_open(int size): returns an address to a shared memory of the specified size
- shared_memory_close(): close the shared memory

Assuming that the system can provide only one shared memory per process with a maximum size equal to one page.

Part 2: Semaphores

Implementation the following system calls for semaphores:

- semaphore semaphore_create(int id): create a semaphore with the provided id
- semahore_wait(semaphore x): wait on the given semaphore
- semaphore_signal(semaphore x): signal the given semaphore
- semaphore_destroy(semaphorex): destroy the given semaphore

Part 3: Dining Philosopher's problem

A solution to the Dining Philosopher's problem that satisfies all the requirements for a solution to a critical section problem.

Shared Memoery

shared_memory_open(int size)

For implemeting this function, we have created a function **int ShareMem(int size)** in **ksyscall.h**.It calls function $kernel- > currentThread- > Attach(size)$.

In $Thread :: Attach$ function,
we do size of the shared memory equals to size declared if it is smaller than max value 128.Here, we have used $No\_pr = ((NumPhysPages - 1)/MaxProc)$ as number of page table entries possible. The last possible position for page table entry is used for shared memory implementation and it returns the address space of size $((No\_pr - 1) * PageSize)$. and we put *SharedMemStatus* is equal to 1 ;
We have developed following features while implementing shared_memory_open() that,

- If more than one process needs more than one size of memory ,we assign each process the memory equal to largest demanded size in bytes.For example, we got demand of 20 pages and also of 50 pages,then in both instance we will provide 50 pages.
- If we create a child process out of parent,it can access the memory shared by parent using the variable ,which stores the address of the shared memory,because it get passed on by parent to child.
- If a process call execute(),the shared memory of process gets detached (For this we have used *Thread::Detach()*)beacuse that space is reintialized and process should again call for shared memory to used it further.
- For each process, it returns the first byte address of the last page of page table,which is not used by any process purpose.

shared_memory_close()

For implemeting this function, we have created **void ShareClose()** in **ksyscall.h**.
It calls function $kernel \rightarrow currentThread \rightarrow Detach()$.
In $Thread :: Detach()$ function,

- We make the last entry of the page table equal to its original value using *Epage*,which is position of the address space of the process in main memory.
- Then using *(* $space \rightarrow RestoreState()$, we restore the state of shared memory. where emphAddrSpace::RestoreState restores the machine state so that this address space can run. For now, tell the machine where to find the page table.

Semaphores

semaphore_create(int id)

For implemeting this function, we have created a function **sem_create(int id)** in **ksyscall.h** It initalizes *mySemaphore* by calling Semaphore(id,1)

- *Semaphore::Semaphore(int Id, int initialValue)*
  Initialize a semaphore, so that it can be used for synchronization.Here we give intial vale to semaphore equal to 1."initialValue" is the initial value of the semaphore.

semahore_wait(semaphore *x)

For implemeting this function, we have created a function **sem_wait(int sem)** in **ksyscall.h**. It calls *mySemaphore* $\rightarrow$*P().*

- Semaphore::P()
  Wait until semaphore value> 0, then decrement. Checking the value and decrementing must be done atomically, so we need to disable interrupts before checking the value.Note that Thread::Sleep assumes that interrupts are disabled when it is called.

semaphore_signal(semaphore *x)

For implemeting this function, we have created a function **sem_signal(int sem)** in **ksyscall.h**. It calls *mySemaphore* $\rightarrow$*V().*

- Semaphore::V()
  Increment semaphore value, waking up a waiter if necessary.As with P(), this operation must be atomic, so we need to disable interrupts. Scheduler::ReadyToRun() assumes that interrupts are disabled when it is called.

semaphore_destroy(semaphore *x)

For implemeting this function, we have created a function **sem_destroy(int sem)** in **ksyscall.h**. It just deletes mySemaphore which has the pointer value of "sem" passed as argument.

Dining Philosopher's problem

We have implemented a solution to the Dining Philosopher's problem.It achieves all the requirements needed for critical section problem in following manner :

- **Mutual Exclusion:**By the definition of semaphore, we know one process modifies the semaphore value, no other process can simultaneously modify the same semaphore. Here we have defined "i" semaphores on "i" chopsticks, so only one chopstick can only be accessed by one philosopher at a time .Hence, we have achieved Mutual Exclusion.

- **Progress:** We have used the following condition that a philosopher can attain his left chopstick only when right chopstick is free.Hence, no condition arrives when each one has one chopstick, so that if someone hungry and chopsticks are free.He has chance to use it. Our sleep time is also different for different philosophers, therefore we can save ourselves from the condition, when each one repeatedly put his left chopsticks down considering right is not available.

Improvements From Last Assignment

- In last assignment, we used to check the case of preemption at each tick,this leads a huge overhead in scheduing algorithm.Now, we have cut down it using the check on preemption after 100 ticks. If we have a thread with higher priority in ReadyList the current thread gets preempted and the higher priority thread is executed first.
- We can now decide the interval of preemption very easily just by changing a macro.
- Now we have simulated the real time I/O device execution.Earlier when a process takes I/O, It gets blocked until the IØis done and we put the thread back in ReadyList(process is till blocked).After that we decrease the sleep time manually and when it is zero.We put the state as READY.

  Now we have changed it so that whenever we call sleep(or IØ) the process gets blocked but does not go in ReadyList. We are handled it using interrupt the thread actually gets in BLOCKED state and after the sleep time over, an interrupt gets invoked and we put the process back in ReadyList with READY state.

  For this we have created a class in **thread.cc** named **SleepTimer**.The class **SleepTimer** implements the abstract class **CallBackObj**.

- Handling the I/O access using interrupt creates a problem that, even when ReadyList get empty, it is possible that some thread is still in block state.For handling this situation, we keep waiting for a process to come in readyList even when readyList is empty, we wait for a specific time.If no interrupt gets generated, we halt the machine.In our machine we have taken that time as ten thousand ticks which goes in idle ticks.So number of idle ticks increases.

- Exit() call this time simple, because now it is purely interrupt driven like real time OS.

Testing

To test Dining Philosopher's problem use assignment1.c which is aved in test folder.