Semaphores in Nachos

Dr. Douglas Niehaus
Michael Jantz
Dr. Prasad Kulkarni

Introduction

- In this lab, we will implement a basic blocking semaphore and a queuing semaphore in Nachos
  - This is a system call based approach, closely following the structure of the pseudo-code given in class for the blocking and queuing semaphores
  - ***Several details in the pseudo-code must be implemented with different mechanisms in Nachos***
- We will use a narrative output from two variations of the *nice_free* program to compare these two implementations.
- Unpack the TAR file that came with this lab, which contains a copy of the Nachos code for this exercise, make it, and tag the source:

```
bash> tar xvzf eecs678_nachos_sem.tar.gz
bash> cd nachos;  make;  ctags -R
```

  - Or, ctags-eR if using Emacs

Critical Section Problem

- Region of code in a process *updating* shared data is called a critical region
- Concurrent updating of share data by multiple processes is dangerous because it can corrupt the contents of shared data
- Critical section problem:
  - How to ensure synchronization between cooperating processes?
- Solution:
  - Lock/Unlock operations ensuring:
    - Mutual Exclusion, Progress, Bounded Wait
- Protocol for solving the critical section problem
  - Request permission to enter critical section
  - Indicate after exit from critical section
  - Only permit a single process at a time

Semaphores

- A solution to the critical section problem
- Definition
  - Access to critical section is controlled by the semaphore
  - Each semaphore abstractly implemented with operations:
    - *init()*, *wait()*, and *signal()*
- Our Model:
  - Each semaphore is a structure in Nachos, containing an integer representing the semaphore state
  - Convention: 0 = Free, 1 = Taken
  - Associate semaphores names in user code with an integer handle used as arguments to semaphore system calls
    - *Attach( )* and *Detach( )* support handles
  - *P( )* implements *Wait( )* and *V( )* implements *signal( )*
    - Semaphore handles used as arguments

Nachos System Level Concurrency

- Nachos simulates a uni-processor system
  - Concurrent processes cannot overlap – only interleave
    - Only thread concurrency not physical
  - Each process runs until it invokes a system call or is interrupted
- Possible Solution: Disable interrupts!
  - Active processes will run without preemption.

```
do {

        disable interrupts;
          critical section
        enable interrupts;
          rest of code

} while(TRUE)
```

- Consider the several limitations of this approach
- Can system calls be made in CS?
- Can it be used in a multi-processor system?
- Should user-code be disabling interrupts?

Nachos System Level Concurrency

- Will not work in multiprocessor systems
  - Processes on different CPUs share data
  - Processes on different CPUs enter CS independently
  - Will work in Nachos: 1 CPU, interrupts cause preemption
- Real hardware provides support of *atomic* instructions
- *Atomic* instructions treated as a single step that cannot be interrupted.
- Consider *TestAndSet* discussed in Chapter 6 materials
- Algorithms presented here require that we ensure mutual exclusion for sections of P/V code
  - Pseudo-code uses P-BW to ensure mutual exclusion in these system calls
  - Nachos mechanism for this policy: *disabling interrupts*
  - Also disables preemption in Nachos because preemption is driven by timer interrupts

Controlling Thread Concurrency in Nachos

- Since preemption of the currently running process in Nachos is instigated by a timer interrupt, we can disable preemption by disabling interrupts
- This is a little crude for a real OS, but fine in Nachos
- One subtle point is that we may not always know if interrupts are already disabled when a subroutine we write is called so:
  - A save-and-disable/restore pattern is often used instead of straight disable/enable, we thus restore the state on entry

```
IntStatus oldLevel;
oldLevel = interrupt->SetLevel(IntOff);   /* disable preemption */
    /* critical section safe from preemption */
(void) interrupt->SetLevel(oldLevel);    /* enable preemption */
```

System Calls in Nachos

- You will be implementing the **PBlock/VBlock** and **PQueue/VQueue** system calls that implement basic blocking and queuing semaphores as discussed in class slides
- User code is in *nachos/test* in *basic_sem_free.c* and *queue_sem_free.c*
  - Note how these programs use **BlockAttach( )** and **QueueAttach( )** to map a semaphore name to a handle
- In Nachos user code, these are library routines implemented in *nachos/userprog/Systemcalls.s*
  - You will not touch the assembler file
- Implementations of these system calls is in C++ in the Nachos code in *nachos/userprog/systemcall.cc*
  - **do_system_call( )** routine takes the SC_* numbers as an argument and maps these onto a call to System_*
  - For example, SC_PBlock -> **System_PBlock( )**

System Calls in Nachos

- Note that **System_PBlock( )** is an empty stub routine
  - As are the other routines you should implement
  - **System_VBlock, System_PQueue, System_VQueue**
- Note that **System_PBlockAttach( )** is not empty
  - Study the attach and detach routines to see how they use their respective tables of semaphore structures
  - Note the **BBSemTable** and **QsemTable** are defined in *nachos/threads/system.cc* where they are also initialized
  - Mapping a semaphore handle to the corresponding semaphore structure pointer is as easy as

```
int handle; /* passed as parameter */
QueueingSemaphore *sem_ptr;
sem_ptr = QSemTable[handle]
```

Sleep and Wakeup Review

- Sleep and wakeup are fundamental services present in all operating systems, although the names vary
- Every OS has a wide variety of reasons to put a process in BLOCKED state (sleep) and to take a sleeping process and wake it up (put it in READY state)
- General interface for this can have a variety of interface routines, but one placing the current thread to sleep will certainly be present
  - Sleep (unique-tag)
  - The argument (unique-tag) is used to uniquely designate a set of processes sleeping for the same reason
- Awakening a process can be done singly or as a group

Sleep and Wakeup Review

- The unique tag is used to distinguish each set of processes in the system sleeping for different reasons
  - Each set is awaiting a specific event
    - Sometimes a single process, sometimes many
  - Unique-tag is often the address of a specific data structure associated with the event
    - In Nachos version the address of a semaphore structure works well
- Sleep and wakeup routines are a collaborative set
  - Hash table with the unique-tag as a key would be suitable
  - Other data structures are certainly possible

Sleep and Wakeup in Nachos

- In nachos/threads/ see *synch.cc* and *synch.h*
  - **SleepWakeupSet** and **SleepWakeupManager**
- **Sleep(tag)** puts current process to sleep
  - Study this code, note set->threads is a List
  - In nachos/threads see *list.h* and *list.cc* for methods
- **Wakeup(tag)** wakes all set members
  - Set of threads can be deleted when empty
- **Wakeup(tag, proc)** wakes a specific member; which one to wake is indicated by (Thread *)**proc**
  - Second level search looks within set of threads associated with the tag to see if a member matching **proc** is present
- You will implement the **Wakeup** routines

Basic Blocking Semaphore Review

- Assumption: P and V are implemented as system calls
  - Could also control *physical* concurrency within the OS if desired, with minor changes to P-BW role
- Applications call them with the address of the semaphore in user address space as an argument
  - Nachos version will use integer handle instead
- OS level global semaphore “busy-flag” available to control concurrent access to CS in P and V code
  - You will change this in the Nachos implementation too
- *sleep*(int event) & *wakeup*(int event) routines block and unblock sets of processes associated with the event
- The *unique-event* routine generates unique tag

Basic Blocking Semaphore Pseudo-Code

```
P-Block(int *S)
{
  int curr-val;
  P-BW(busy-flag);

  curr-val = get-user-var(S);
  while ( curr-val == 1 ) {
    V-BW(busy-flag);
    sleep(unique-event(S));
    P-BW(busy-flag);
    curr-val = get-user-var(S);
  }

  set-user-value(S, 1);
  V-BW(busy-flag);
}
```

```
V-Block(int *S)
{
  P-BW(busy-flag);

  set-user-var(S, 0);
  wakeup(unique-event(S));

  V-BW(busy-flag);
}
```

Basic Blocking Semaphore Implementation

- Now you should be able to get the basic blocking versions of the system calls working
  - Use disable/enable preemption as the mechanism for mutual exclusion from P/V critical sections which is done using P/V-BW in the pseudo-code
  - Note the system call argument is a semaphore handle rather than a user space integer pointer and map it to semaphore in Nachos OS address space
  - With a pointer to the semaphore in OS space you will not need to use get-user-value or set-user-value to operate across OS/User address space boundary
  - You will need to implement the Wakeup(tag) call
  - You test it using *nachos/test/basic_sem_free.c*

bash> cd nachos; ./userprog/nachos -d sem -x ./test/basic_sem_free

Queuing Semaphore

- Assume the semaphore structure is kept in OS address space
  - OS needs access to the queue
- This implies the application uses a "handle" to refer to the semaphore
  - Some form of "attach" or "open" operation for semaphores thus needed
- OS semaphore structure must maintain the queue
  - Note that in *nachos/threads/synch.cc* that the **QueueingSemaphore** class inherits from the blocking semaphore and just adds a queue
- Wakeup-proc in the pseudo-code wakes a specific process

Queuing Semaphore

```
P-Queue(int S-ref)
{
  semaphore-t *S;
  S = sem-hdl-to-ptr(S-ref);

  P-BW(busy-flag);
  if ( S->value == 1 ) {
    enqueue(curr-proc, S);
    V-BW(busy-flag);
    sleep(unique-event(S));
  } else {
    S->value = 1;
    V-BW(busy-flag);
  }
}
```

```
V-Queue(int S-ref)
{
  semaphore-t *S;
  pcb-t       *proc;

  S = sem-hdl-to-ref(S-ref);
  P-BW(busy-flag);
  if ( (proc = dequeue(S)) != (pcb-t *)0 ) {
    wakeup-proc(proc, unique-event(S));
  } else {
    S->value = 0;
  }
  V-BW(busy-flag);
}
```

Queuing Semaphore Implementation

- Now you should be able to get the queuing versions of the system calls working
  - Use disable/enable preemption as the mechanism for mutual exclusion from P/V critical sections which is done using P/V-BW in the pseudo-code
  - Note the system call argument is a semaphore handle as the pseudo-code assumes
  - You will need to implement the Wakeup(tag,proc) call
  - You test it using *nachos/test/queue_sem_free.c*

bash> cd nachos; ./userprog/nachos -d sem -x ./test/queue_sem_free

- Interpreting the output can be fairly subtle as it narrates a fairly large series of events

Testing Your Solutions

- The two example programs, *basic_sem_free* and *queue_sem_free*, are similar in structure:
  - Each program forks four child processes and prints a different character in each child process (similar to *nice_free*).
- Character printing in each process is controlled by two loops which iterate INNER_LOOP_LIMIT and OUTER_LOOP_LIMIT
- Each process locks a semaphore, SEM_ONE, shared among the processes before proceeding to the inner loop
- A correct implementation will show that no process is interleaved while printing characters inside its inner loop
  - Characters for each process will thus appear in block of size INNER_LOOP_LIMIT
  - Both semaphore implementations should give same behavior

Testing Your Solutions

- Run the two programs from the *nachos* directory using:
  bash> ./userprog/nachos -d sem -x ./test/basic_sem_free
  bash> ./userprog/nachos -d sem -x ./test/queue_sem_free
- These will generate a long narrative output.
- Each type of line in the narrative is explained on the following slides
- A quick check is that you see patterns such as that to the right
- This indicates A was able to acquire the semaphore BB-S1 and hold it for the duration of its inner loop.

```
BBAcquire(T-A, BB-S1)
CS(T-A -> T-C)
BBSleep(T-C, BB-S1)
CS(T-C -> T-B)
BBSleep(T-B, BB-S1)
CS(T-B -> T-D)
BBAttemptToLock(T-D, BB-S1)
BBSleep(T-D, BB-S1)
CS(T-D -> T-A)
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA
BBRelease(T-A, BB-S1)
BBWakeupAll(T-A, BB-S1)
```

Output Narration Components

- This is provided as a reference of what each type of line in the output narration means
- The code emitting the narrative is already in the Nachos code

NameThread("T-2" -> "T-A"):	Assigned name "T-A" to thread "T-2"
CS(T-A -> T-D):	Context switch from "T-A" to "T-D"
BBAttach(T-A, “BB-S1”):	Attach"T-A" to basic blocking semaphore BB-S1
BBDetach(T-A, “BB-S1”):	Detach "T-A" from basic blocking semaphore BB-S1
QAttach(T-A, “Q-S1”):	Attach thread "T-A" to queuing semaphore Q-S1
QDetach(T-A, “Q-S1”):	Detach thread "T-A" from queuing semaphore Q-S1

Output Narration

- These outputs should be produced by DEBUG statements in the code you write. The relevant DEBUG lines are in the stubs

BBAttemptToLock(T-A, BB-S1): T-A attempt to lock BB semaphore "BB-S1" in PBlock()

BBSleep(T-A, BB-S1): T-A sleeps waiting on BB semaphore "BB-S1" in PBlock()

BBAcquire(T-A, BB-S1): T-A acquired BB semaphore "BB-S1" in PBlock()

BBRelease(T-A, BB-S1): T-A released BB semaphore "BB-S1" in VBlock()

BBWakeupAll(T-A, BB-S1): T-A awakened all threads waiting on BB-S1 because T-A released it in VBlock()

QAttemptToLock(T-A, Q-S1): T-A attempted to lock queuing semaphore “Q-S1" in PQueue()

QSleep(T-A, Q-S1): T-A sleeps waiting on queuing semaphore "Q-S1" in PQueue()

QAcquire(T-A, Q-S1): T-A acquired queuing semaphore "Q-S1" in PQueue()

QRelease(T-A, Q-S1): T-A released queuing semaphore "Q-S1" in VQueue()

QWakeupProc((T-A, Q-S1) --WOKE-> T-B): T-A awoke T-B waiting on "Q-S1" T-A it in VQueue()

Final Verification

- When you are through with your implementation:
  - Verify your solution is correct by analyzing output of both programs using the description of the narrative output statements provided
  - Answer the questions posed on the lab website

Conclusions

- At some levels this exercise was simple, but several important points are present and should be appreciated
  - You have mapped the abstract algorithms for two types of binary semaphores into a specific environment
  - You have seen that basic algorithmic policy goals such as sleep/wakeup and mutual exclusion in the semaphore code (P-BW) can be achieved by different mechanisms in different environments
  - You have seen the relationship that exists between the User-level part of a system call and the OS-level part, and have seen how that relationship is implemented in the context of Nachos
  - You have seen that use of concurrency control can substantially change the behavior of programs and that this aspect needs to be considered as well as those more obvious