CS 350 - Operating Systems **Winter 2018**

Lecture 5: January 17, 2018

Lecturer: Lesley Istead *Notes By: Harsh Mistry*

5.1 Synchronization Continued

5.1.1 Semaphore

- A semaphore is a synchronization primitive hat can be used to enforce mutual exclusion requirements. It can also be used to solve other kinds of synchronization problems.
- A semaphore is an object that has an integer value, and that supports two operations:

P: if the semaphore value is greater than 0, decrement the value. Otherwise, wait until the value is greater than 0 and then decrement it.

V: increment the value of the semaphore

5.1.2 Condition Variables

- Each condition variable is intended to work together with a lock: condition variables are only used from within the critical section that is protected by the lock
- Three operations are possible on a condition variable

Wait: This causes a thread to block, and it releases the lock associated with the condition variable. Once the thread is unblocked in reacquires the lock

Signal: If threads are blocked on the signaled condition variable, then one of those threads is unblocked.

Broadcast: Like signal, but unblocks all threads that are blocked on the condition variable.

5.1.2.1 Using Condition Variables

- Condition variables get their name because they allow threads to wait for arbitrary conditions to become true inside of a critical section.
- Normally, each condition variable corresponds to a particular condition that is of interest to an application.
- when a condition is not true, a thread can wait on the corresponding condition variable until it becomes true
- when a thread detects that a condition is true, it uses signal or broadcast to notify any threads that may be waiting

5.1.2.2 Waiting on Condition Variable

- when a blocked thread is unblocked (by signal or broadcast), it reacquires the lock before returning from the wait call
- a thread is in the critical section when it calls wait, and it will be in the critical section when wait returns. However, in between the call and the return, while the caller is blocked, the caller is out of the critical section, and other threads may enter.
- In particular, the thread that calls signal (or broadcast) to wake up the waiting thread will itself be in the critical section when it signals. The waiting thread will have to wait (at least) until the signaller releases the lock before it can unblock and return from the wait call.

5.1.3 Deadlocks

Deadlocks when multiple threads try to acquire resources that they each other already hold.

5.1.3.1 Two Techniques for Deadlock Prevention

- No Hold and Wait : Prevent a thread from requesting resources if it currently has resources allocated to it. A thread may hold several , but to do so it must make a signal request for all of them.
  Hold and Wait Implementation (Possible Exam Question)

```
1 Bool Try_Acquire (lock)
2   lock available ?
3     YES -> take lock -> return true
4     NO -> return false
```

  Sample Usage of Hold and Wait (Possible Exam Question)

```
1 acquire(A)
2 while(! try_acquire(B)){
3   release(A);
4   acquire(A);
5 }
```

- Order the resources types, and require that each thread acquire resources in increasing resource type order. That is, a thread may make no requests for resources in increasing resource type order. That is a thread my make no requests for resources of type less than or equal i if it is holding reosurces of type i

5.2 Processes and System Calls

5.2.1 Processes

Definition 5.1 *A process is an environment in which an application program runs.*

- A process includes virtualized resources that its program can use
- Processes are created and managed by the kernel
- Each programs process isolates its from other programs in other processes

5.2.2 System Calls

- System calls are the interface between processes and the kernel.
- A process uses system calls to request operating system services.
- (i.e `fork, execv, waitpid, getpid`)

5.2.3 System Calls for Process Management

- `fork` creates a new process (a child) that is a clone of the original
  - after fork, both parent and child are executing copies of the same program
  - virtual memories of parent and child are identical at the time of the fork, but may diverge afterwards
  - fork is called by the parent, but returns in both the parent and the child
  - parent and child see different return values from fork
- `_exit` terminates the process that call its and possibly supplies a exit status code that is recorded by the kernel
- `waitpid` lets a process wait for anotehr to terminate, and retrieve its exit status code.

5.2.4 The execv System Call

- `execv` changes the program that a process is running
- The calling process's current virtual memory is destroyed
- The process gets a new virtual memory, initialized with the code and data of the new program to run
- After `execv`, the new program starts executing.