OPERATING SYSTEMS

Synchronization Examples

Suresh Jamadagni

Department of Computer Science

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Synchronization Examples

- Windows
- Linux
- Pthreads
- Solaris

Suresh Jamadagni

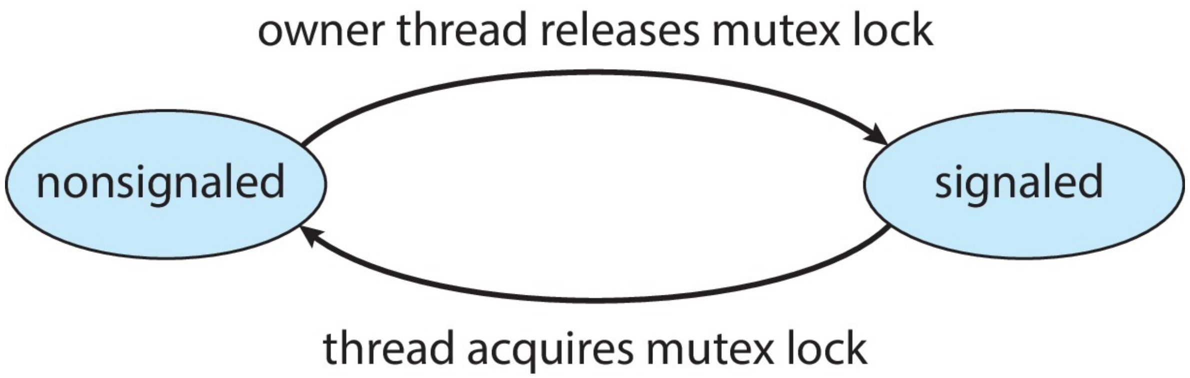
Department of Computer Science

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - For reasons of efficiency, kernel ensures that a thread will never be preempted while holding a spinlock.
- Also provides **dispatcher objects** outside the kernel, to synchronize mutex locks, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable (i.e notify a waiting thread when a desired condition occurs)
  - Timers notify one or more thread when time expired

Windows Synchronization - Mutex dispatcher object

- Dispatcher objects may be in either a **signaled-state (**object available and a thread will not block) or a **non-signaled state** (object not available, thread will block)
- A Relationship exists between the state of a dispatcher object and the state of a thread.
  - State of a thread changes from ready to waiting and vice-versa



Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization (Cont.)

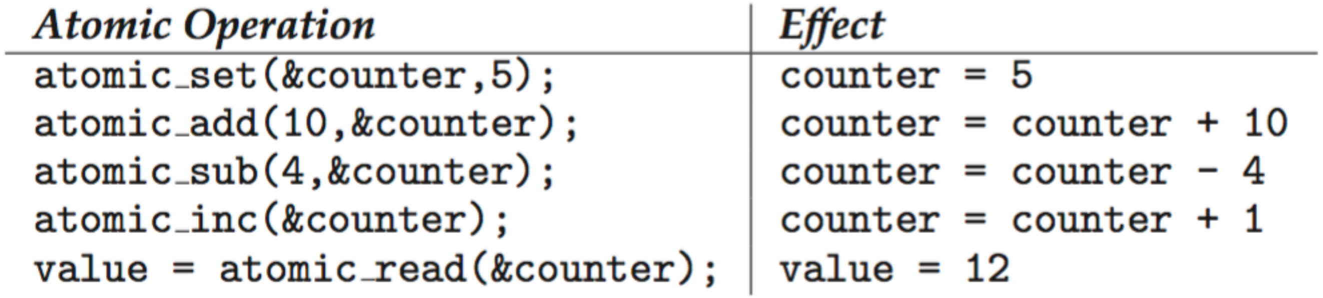
- Atomic variables

  atomic_t is the data type for atomic integer

- Consider the variables

  atomic_t counter;

  int value;



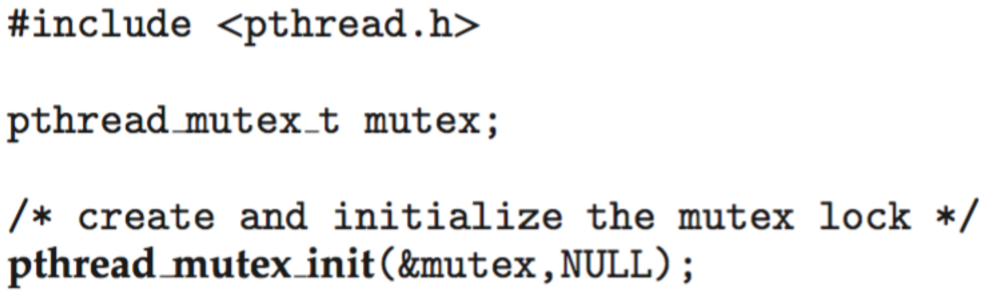
Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Pthreads (POSIX) Synchronization

- Pthreads API is OS-independent, widely used on UNIX, Linux, and macOS
- It provides:
  - mutex locks
  - semaphores
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

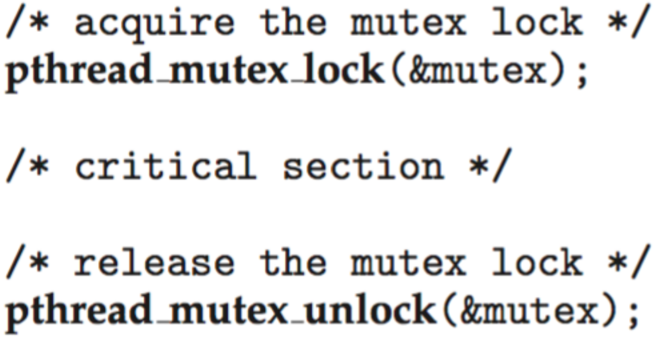
POSIX Mutex Locks

- Creating and initializing the lock



```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring



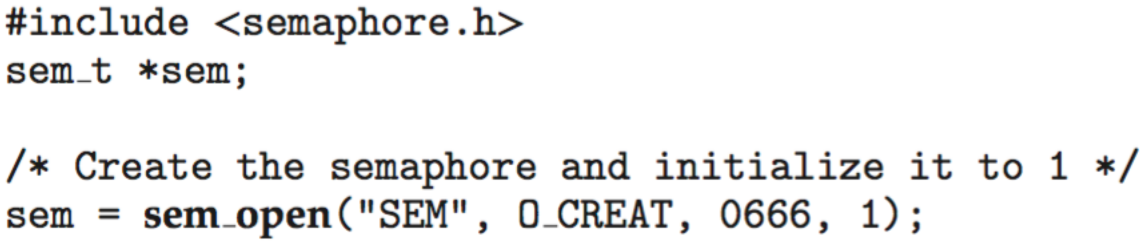
```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores (have actual names in the file system) can be shared by multiple unrelated processes
- Unnamed semaphores can be used only by threads belonging to the same process.

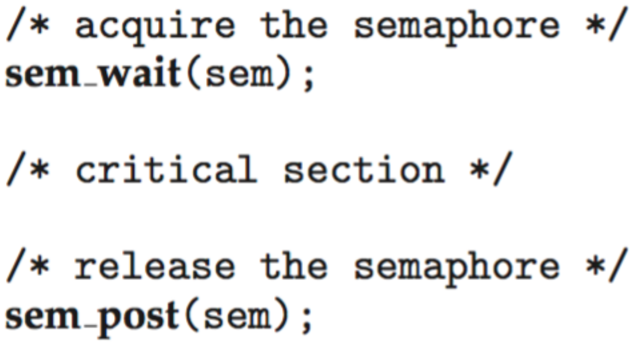
POSIX Named Semaphores

- Creating and initializing the semaphore:



```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

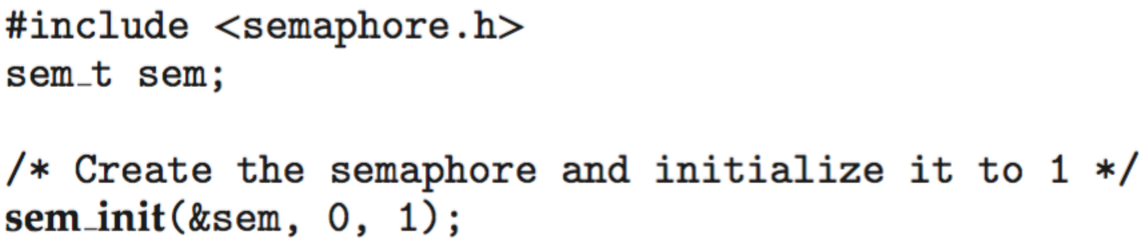
- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:



```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

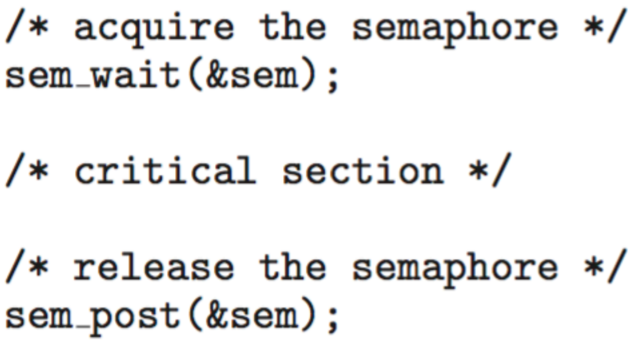
POSIX Unnamed Semaphores

- Creating and initializing the semaphore:



```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

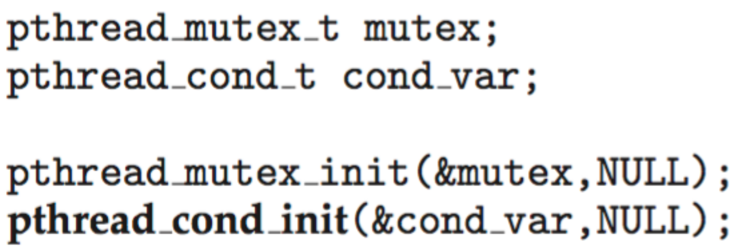
- Acquiring and releasing the semaphore:



```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Condition Variables

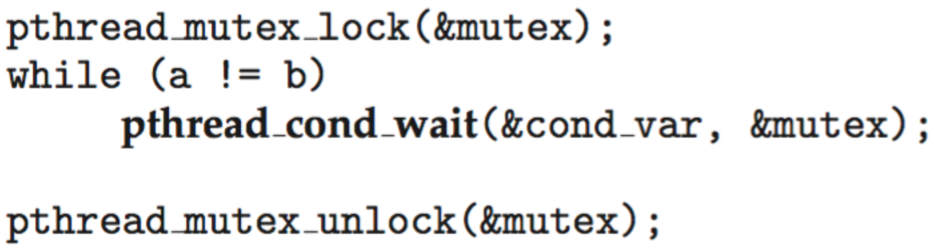
- Since POSIX is typically used in C/C++ and these languages do not provide a monitor (A high-level abstraction that provides a convenient and effective mechanism for process synchronization) , POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:



```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

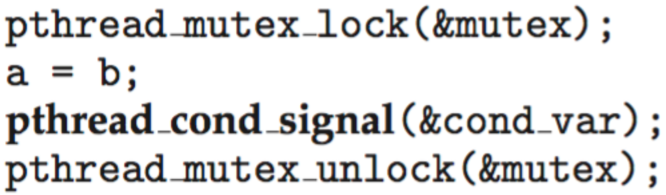
POSIX Condition Variables

- Thread waiting for the condition **a == b** to become true:



```
pthread_mutex_lock(&mutex);
while (a != b)
      pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:



```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments (< a few 100 instructions)
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread (i.e. the thread holding the lock is not currently in run state), block and sleep waiting for signal of lock being released

Solaris Synchronization (Cont.)

- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
    - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

PES UNIVERSITY ONLINE

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu