Operating Systems

Xinu - Coordination of Concurrent Processes

1

Coordination and Synchronization

- Concurrent processes need to cooperate when sharing global data
  - Mutual exclusion accessing variables
- Allow a subset of processes to contend for access to a resource
  - Do not disable all interrupts
  - Block only appropriate processes
- Fair access
  - Each contending process eventually receives access
  - No starvation

2

Semaphores

- *wait* and *signal* operations
  - also known as down/up or P/V
- Provide a solution for
  - Mutual exclusion
    - Binary semaphore
  - Producer-consumer interaction
    - Counting semaphore
- Busy waiting
  - *Generally* to be avoided as it deprives other processes of the CPU
  - Sometimes necessary for correctness
  - Sometimes desirable for performance (must be used judiciously)

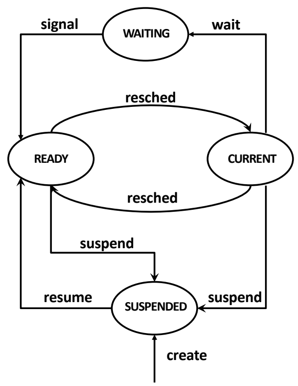
3

Scheduler Interaction

- Semaphores without busy waiting involve process lists and management by the scheduler
- When a semaphore is signaled, the scheduler should activate a waiting process (if the list is not empty)
- Scheduler decision
  - Priority? Wait time? Random?
- Random works but requires random number generation
- Xinu uses FCFS

4

Process State: Waiting (PR_WAIT)



5

Semaphore Structures

```
/* semaphore.h - isbadsem */

#ifndef NSEM
#define NSEM       120 /* number of semaphores, if not defined  */
#endif

/* Semaphore state definitions */

#define S_FREE  0       /* semaphore table entry is available    */
#define S_USED  1       /* semaphore table entry is in use       */

/* Semaphore table entry */
struct  sentry  {
    byte    sstate;      /* whether entry is S_FREE or S_USED    */
    int32   scount;      /* count for the semaphore              */
    qid16   squeue;      /* queue of processes that are waiting  */
                         /*  on the semaphore                    */
};

extern  struct  sentry semtab[];  /* index identifies semaphore  */

#define isbadsem(s) ((int32)(s) < 0 || (s) >= NSEM);
```

6

Wait system call - wait.c

```
/* wait.c - wait */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wait  -  Cause current process to wait on a semaphore
 *------------------------------------------------------------------------
 */
syscall wait(  sid32   sem )        /* semaphore on which to wait   */
{
    intmask mask;                   /* saved interrupt mask         */
    struct  procent *prptr;        /* ptr to process' table entry   */
    struct  sentry *semptr;        /* ptr to sempahore table entry  */

    mask = disable();

    if (isbadsem(sem)) {
        restore(mask);
        return SYSERR;
    }
```

7

```
    semptr = &semtab[sem];
    if (semptr->sstate == S_FREE) {
        restore(mask);
        return SYSERR;
    }

    if (--(semptr->scount) < 0) {        /* if caller must block    */
        prptr = &proctab[currpid];
        prptr->prstate = PR_WAIT;         /* set state to waiting    */
        prptr->prsem = sem;               /* record semaphore ID     */
        enqueue(currpid,semptr->squeue); /* enqueue on semaphore    */
        resched();                        /*   and reschedule        */
    }

    restore(mask);
    return OK;
}
```

- Once enqueued, a process remains in PR_WAIT until it reaches the head of the queue and another process signals the semaphore
- Then it is moved to the ready queue
- ctxsw -> resched -> wait -> process code

8

Signal system call - signal.c

```
/* signal.c - signal */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  signal  -  Signal a semaphore, releasing a process if one is waiting
 *------------------------------------------------------------------------
 */
syscall signal( sid32  sem )          /* id of semaphore to signal     */
{
    intmask mask;                     /* saved interrupt mask           */
    struct  sentry *semptr;           /* ptr to sempahore table entry  */

    mask = disable();
    if (isbadsem(sem)) {
        restore(mask);
        return SYSERR;
    }
```

9

```
    semptr= &semtab[sem];

    if (semptr->sstate == S_FREE) {
        restore(mask);
        return SYSERR;
    }

    if ((semptr->scount++) < 0) {         /* release a waiting process */
        ready(dequeue(semptr->squeue), RESCHED_YES);
    }

    restore(mask);
    return OK;
}
```

- A non-negative semaphore count means that queue is empty
  - Count of N means that wait can be called N times before a process blocks
- A semaphore count of negative N means that the queue contains N waiting processes

10

Static and Dynamic Allocation

- Static allocation – fixed set of semaphores at system compile time
- Dynamic allocation – the system includes functions for creating and destroying semaphores
- Static allocation
  - Reduction in overhead
  - Reduction in flexibility
- Most systems provide dynamic allocation as we have seen

11

Dynamic semaphore allocation

```
/* semcreate.c - semcreate, newsem */
#include <xinu.h>

local   sid32   newsem(void);
/*------------------------------------------------------------------------
 *  semcreate  -  create a new semaphore and return the ID to the caller
 *------------------------------------------------------------------------
 */
sid32   semcreate(  int32  count       /* initial semaphore count*/)
{
    intmask mask;                     /* saved interrupt mask   */
    sid32   sem;                      /* semaphore ID to return */

    mask = disable();

    if (count < 0 || ((sem=newsem())==SYSERR)) {
        restore(mask);
        return SYSERR;
    }
    semtab[sem].scount = count;        /* initialize table entry */

    restore(mask);
    return sem;
}
```

12

```
/*------------------------------------------------------------------------
 *  newsem  -  allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
local   sid32   newsem(void)
{
    static  sid32   nextsem = 0;   /* next semaphore index to try    */
    sid32   sem;                   /* semaphore ID to return  */
    int32   i;                     /* iterate through # entries  */

    for (i=0 ; i<NSEM ; i++) {
        sem = nextsem++;
        if (nextsem >= NSEM)
            nextsem = 0;
        if (semtab[sem].sstate == S_FREE) {
            semtab[sem].sstate = S_USED;
            return sem;
        }
    }
    return SYSERR;
}
```

13

```
/* semdelete.c - semdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 * semdelete  --  Delete a semaphore by releasing its table entry
 *------------------------------------------------------------------------
 */
syscall semdelete( sid32   sem  )   /* ID of semaphore to delete    */
{
    intmask mask;                   /* saved interrupt mask          */
    struct  sentry *semptr;         /* ptr to semaphore table entry  */

    mask = disable();
    if (isbadsem(sem)) {
        restore(mask);
        return SYSERR;
    }

    semptr = &semtab[sem];
    if (semptr->sstate == S_FREE) {
        restore(mask);
        return SYSERR;
    }
```

14

```
    semptr->sstate = S_FREE;

    while (semptr->scount++ < 0) {    /* free all waiting processes */
        ready(getfirst(semptr->squeue), RESCHED_NO);
    }
    resched();
    restore(mask);
    return OK;
}
```

- Makes all waiting processes ready
- Deleting a semaphore with waiting processes could be considered an error

15

```
/* semreset.c - semreset */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  semreset  --  reset a semaphore's count and release waiting processes
 *------------------------------------------------------------------------
 */
syscall semreset(
      sid32     sem,            /* ID of semaphore to reset  */
      int32     count           /* new count (must be >= 0)  */
    )
{
    intmask mask;               /* saved interrupt mask      */
    struct  sentry *semptr;     /* ptr to semaphore table entry */
    qid16   semqueue;           /* semaphore's process queue ID */
    pid32   pid;                /* ID of a waiting process   */

    mask = disable();

    if (count < 0 || isbadsem(sem) || semtab[sem].sstate == S_FREE) {
        restore(mask);
        return SYSERR;
    }
```

16

```
    semptr = &semtab[sem];

    semqueue = semptr->squeue;      /* free any waiting processes */
    while ((pid=getfirst(semqueue)) != EMPTY)
        ready(pid,RESCHED_NO);

    semptr->scount = count;         /* reset count as specified */
    resched();
    restore(mask);
    return(OK);
}
```


Coordination across Multiple Processors / Cores

- Xinu semaphores as described work well on single CPUs
- Disabling interrupts on a system with multiple cores is inefficient
  - No progress on I/O on any core, and no timer interrupts!
- Test and set instruction can be used for atomic access without disabling interrupts
  - Or compare and swap
- The book describes spin locks in this context, but atomic instructions can be used with waiting and queues as well
  - Adaptive or hybrid solutions are possible