6 Process Coordination

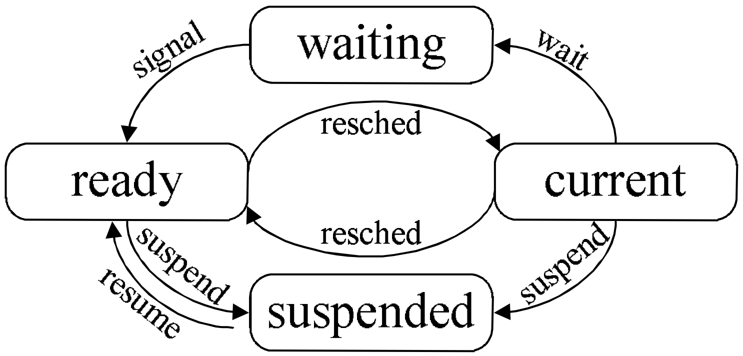
- Process coordination is required to **synchronize the actions of processes** and to **cooperate in sharing resources**.
- In the first Chapter we discussed the **producer-consumer-problem** and **mutual exclusion**.
- The simplest way to synchronize process coordination is provided by **semaphores**:
  - Each semaphore s consists of an **integer count**. The system call `wait(`s`)` is used to **decrement** s, system call `signal(`s`)` is used to **increments** s.
  - If s becomes **negative** when a process executes `wait(`s`)`, that process is **delayed**. Each time `signal(`s`)` is called **one delayed process is ready to run again**.

6.1 Low-level Coordination Techniques

- XINU procedures `ready(`s`)` and `resume(`s`)` employ code for process coordination through the **disabling of interrupts**, because when the process table needs to be updated **only one process** is allowed to do so at every moment.
- However, disabling interrupts has an undesirable effect: it **all but one process**, and **limits** what that process can do (e.g., I/O operations). We need process coordination primitives enabling the coordination of a given subset of processes **without stopping the other processes**, and **without limiting what these processes can do**.

6.2 Implementation of High-Level Coordination Primitives

- When a process needs to **wait** for some condition to become true, the system places it on a **list associated with semaphore** s. So each semaphore has its own **list of waiting processes**.
- To **delay the current process**, `wait(`s`)` enqueues it on the list for s, and calls `resched` allowing another process to run.
- `signal(`s`)` checks that list whenever it is called. When the list of waiting processes associated with semaphore s is **nonempty**, signal **reactivates exactly one process** by moving it back to the ready list.
- This leads to a new process state called `PRWAIT`:



6.2.1 Semaphore Data Structures

- Semaphore info is kept in the **global semaphore table** `SEMAPH`, initialized at system initialization and consisting of `sentry` structures:

```
/* sem.h - isbadsem */

#ifndef NSEM
#define NSEM            45      /* number of semaphores, if not defined */
#endif

#define SFREE   '\01'           /* this semaphore is free               */
#define SUSED   '\02'           /* this semaphore is used               */

struct  sentry  {               /* semaphore table entry                */
        char    sstate;         /* the state SFREE or SUSED             */
        short   semcnt;         /* count for this semaphore             */
        short   sqhead;         /* q index of head of list              */
        short   sqtail;         /* q index of tail of list              */
};
extern  struct  sentry  semaph[];
extern  int     nextsem;

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

Implementation of High-Level Coordination Primitives

- `wait(`s`)` is a system procedure courtesy of the interface of XINU. It **decreases** the count of s. If it remains **nonnegative** (≥ 0) wait **returns to the caller**. Otherwise, the calling process is enqueued on the list for s, its state is changed to `PRWAIT`, and `resched` is called to switch to a ready process.
- The call to `signal(`s`)` moves one waiting process back to the ready list, it becomes eligible for CPU use, and is eventually executed.
- From the point of the waiting process, its last action was calling `ctxsw`. Then this call to `ctxsw` returns to `resched`, that returns to `wait`, and that returns to where `wait` had been called!
- A nonnegative semaphore count means the queue is empty, a semaphore count of negative n means the queue contains n waiting processes.

High-Level Coordination: `wait`

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 * wait  --  make current process wait on a semaphore
 *------------------------------------------------------------------------
 */
SYSCALL wait(sem)
        int     sem;
{
        char    ps;
        register struct sentry  *sptr;
        register struct pentry  *pptr;

        disable(ps);
        if (isbadsem(sem) || (sptr= &semaph[sem])->sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        if (--(sptr->semcnt) < 0) {
                (pptr = &proctab[currpid])->pstate = PRWAIT;
                pptr->psem = sem;
                enqueue(currpid,sptr->sqtail);
                resched();
        }
        restore(ps);
        return(OK);
}
```

High-Level Coordination: `signal`

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 * signal  --  signal a semaphore, releasing one waiting process
 *------------------------------------------------------------------------
 */
SYSCALL signal(sem)
register int    sem;
{
        register struct sentry  *sptr;
        char    ps;

        disable(ps);
        if (isbadsem(sem) || (sptr= &semaph[sem])->sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        if ((sptr->semcnt++) < 0)
                ready(getfirst(sptr->sqhead), RESCHYES);
        restore(ps);
        return(OK);
}
```

Semaphore Creation and Deletion

- The system **preallocates** the head and tail nodes **for each semaphore list** in the *q*-structure at system initialization time. System calls `screate` and `sdelete` **allocate and delete semaphores**.

```
/* screate.c - screate, newsem */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 * screate  --  create and initialize a semaphore, returning its id
 *------------------------------------------------------------------------
 */
SYSCALL screate(count)
        int     count;                  /* initial count (>=0)          */
{
        char    ps;
        int     sem;

        disable(ps);
        if ( count<0 || (sem=newsem())==SYSERR ) {
                restore(ps);
                return(SYSERR);
        }
        semaph[sem].semcnt = count;
        /* sqhead and sqtail were initialized at system startup */
        restore(ps);
        return(sem);
}
```

Semaphore Creation and Deletion: `newsem`

```
/*------------------------------------------------------------------------
 * newsem  --  allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
LOCAL   newsem()
{
        int     sem;
        int     i;

        for (i=0 ; i<NSEM ; i++) {
                sem=nextsem--;
                if (nextsem < 0)
                        nextsem = NSEM-1;
                if (semaph[sem].sstate==SFREE) {
                        semaph[sem].sstate = SUSED;
                        return(sem);
                }
        }
        return(SYSERR);
}
```

Semaphore Creation and Deletion: `sdelete`

```
/* sdelete.c - sdelete */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 * sdelete  --  delete a semaphore by releasing its table entry
 *------------------------------------------------------------------------
 */
SYSCALL sdelete(sem)
        int     sem;
{
        char    ps;
        int     pid;
        struct  sentry  *sptr;          /* address of sem to free       */

        disable(ps);
        if (isbadsem(sem) || semaph[sem].sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        sptr = &semaph[sem];
        sptr->sstate = SFREE;
        if (nonempty(sptr->sqhead)) {   /* free waiting processes       */
                while( (pid=getfirst(sptr->sqhead)) != EMPTY)
                        ready(pid,RESCHNO);
                resched();
        }
        restore(ps);
        return(OK);
}
```