6 Process Coordination

Process coordination is required

- to synchronize process' actions
- and to cooperate in sharing resources

In Chapter I : producer-consumer-problem, mutual exclusion
The simplest way to synchronize process coordination is provided by **semaphores**

Semaphores

- Each semaphore s consists of an integer count.
- System call *wait(s)* is used to decrements s
- System call *signal(s)* is used to increments s
- If s becomes negative when a process executes *wait(s)*, that process is delayed
- A delayed process is ready to run again each time *signal(s)* is called

6.1 Low-Level Coordination

- *ready* and *resume* disable process interrupts when they need to coordinate.
- Disabling interrupts is, i.g., not adequate :
  - it stops all but one process, and limits what that process can do
- Disabling interrupts has also an undesirable global effect.

6.2 Implementation Of High-Level Coordination Primitives

- The Xinu implementation of counting semaphores avoids busy waiting by adding a new process state called **PRWAIT**
- Each semaphore has its own list of waiting processes

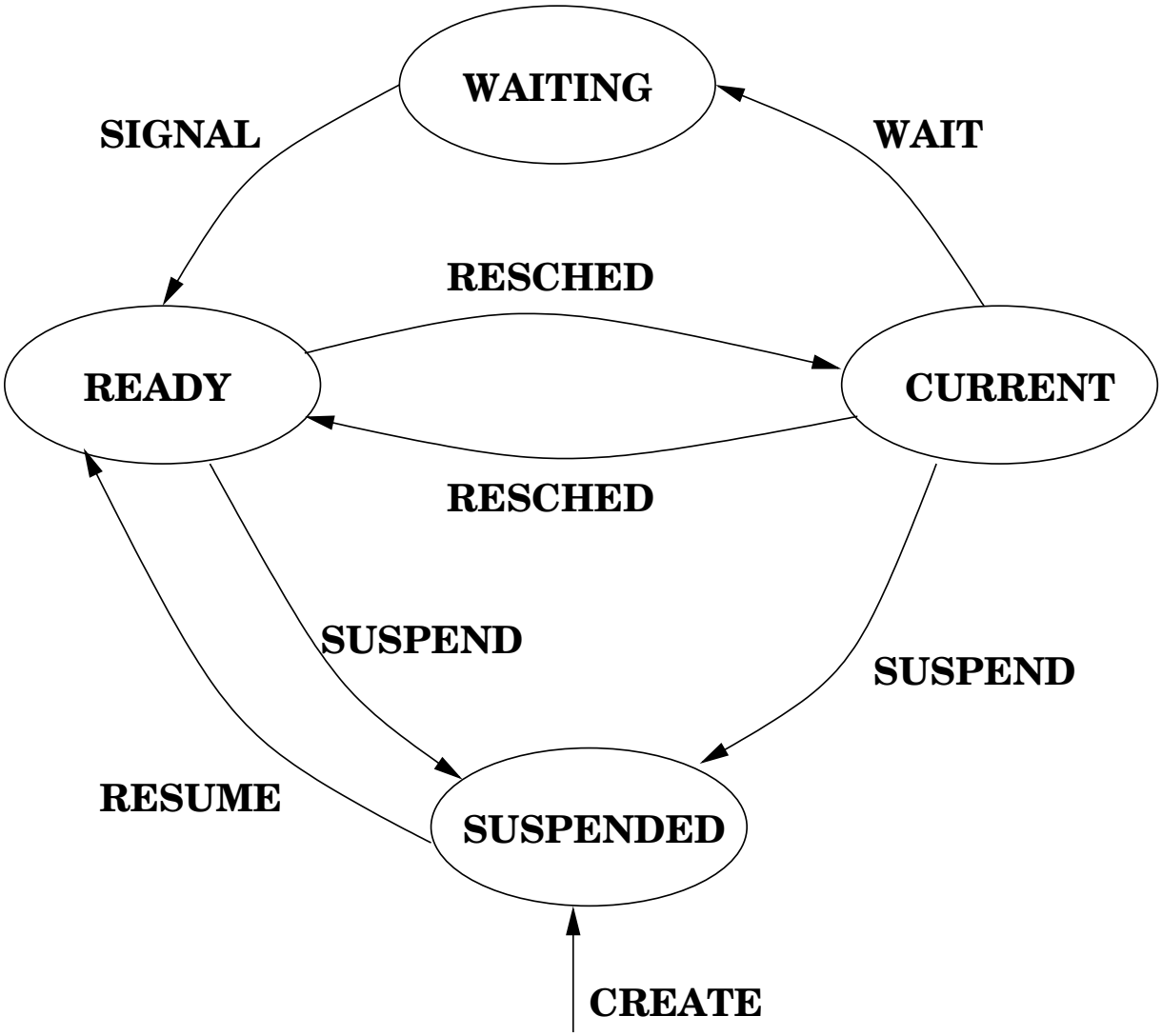

Figure 6.1: Transitions for the waiting state

6.3 Semaphore Data Structures

```
#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)
```

Semaphores are identified by their index in the global semaphore table **semaph**

System Call – wait

```
SYSCALL wait(sem)
    ...         {   ....
        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);
```

System Call – signal

```
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);  }
```

6.4 Semaphore Creation and Deletion

semaphore creation

```
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);
}
```

```
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 deletion

```
SYSCALL sdelete(sem)
    ....    {  .....
        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);
}
```