PART 5

Process Coordination
And Synchronization
Location Of Process Coordination
In The Xinu Hierarchy
Coordination Of Processes

- Necessary in a concurrent system
- Avoids conflicts when accessing shared items
- Allows processes to cooperate
- Can be used when
  - Process waits for I/O
  - Process waits for another process
- Example of cooperation among processes: UNIX pipes
Two Approaches To Process Coordination

- Use facilities supplied by hardware
- Use facilities supplied by the operating system

Note: we will focus on latter
Important Problems That Process Coordination Mechanisms Solve

- Mutual exclusion
- Producer/consumer interaction
Mutual Exclusion Problem

- Concurrent processes access shared data
- Nonatomic operations can produce unexpected results
- Example: multiple steps used to increment variable $z$
  - Load variable $z$ into register $i$
  - Increment register $i$
  - Store register $i$ in variable $x$
Illustration Of Two Processes Attempting To Increment A Shared Variable Concurrently

- process_1
  - load reg_1, x
  - incr reg_1
    - interrupt occurs (context switch)
  - store x, reg_1

- process_2
  - load reg_1, x
  - incr reg_1
  - store x, reg_1
    - interrupt occurs (context switch)
To Prevent Problems

- Insure that only one process accesses a shared item at any time
- Trick: once a process obtains access, make all other processes wait
- Two solutions
  - Test-and-set (implemented in hardware)
  - Semaphores (implemented in software)
Handling Mutual Exclusion With Hardware
(Using The Test-And-Set Instruction)

- Atomic hardware operation, \texttt{tset}, tests whether a memory location is zero and sets it to nonzero

- Initialization (or to declare the shared item is not in use): set memory location to zero

\[ m = 0; \]

- To obtain access, execute the following loop:

\[
\text{while } \text{tset}(m) \\
\quad ; /* \text{do nothing} */
\]

- Only one process can access at any time

- Approach known as \textit{busy waiting}

- Used in multiprocessors
Handling Mutual Exclusion With Software
(Using An Operating System Facility)

• Operating System
  – Supplies abstraction to control mutual exclusion
  – Mechanism used to protect a given shared object is known as a \textit{mutex} facility
  – Guarantees only one process passes a mutex at any time

• Applications must be programmed to use mutex mechanisms

• Unlike test-and-set, mutex implementations avoid busy waiting
Terminology For Mutual Exclusion

- Each item of shared data must be protected from concurrent access
- Calls to OS functions inserted in code
  - Before access to shared data
  - After access to shared data
- Protected code known as *critical section*
- OS ensures at most one process executes critical section at any time
At what level of granularity should mutual exclusion be applied in an operating system?
Low-Level Mutual Exclusion

- Mutual exclusion needed
  - By application processes
  - Inside operating system
- Mutual exclusion can be guaranteed provided no context switching occurs
- Context changed by
  - Interrupts
  - Calls to `resched`
- Low-level mutual exclusion: mask interrupts and avoid rescheduling
Interrupt Mask

- Hardware mechanism that controls interrupts
- Internal register; may be part of processor status word
- Typically, zero value means interrupts can occur
- OS can
  - Examine current interrupt mask (find out whether interrupts are enabled)
  - Set interrupt mask to allow or prevent interrupts
Masking Interrupts

• Important principle:

No operating system function should contain code to explicitly enable interrupts.

• Technique used: given function
  – Saves current interrupt status
  – Disables interrupts
  – Proceeds through critical section
  – Restores interrupt status from saved copy

• Important idea: allows nested calls
Why Interrupt Masking Is Insufficient

- It works! But...

- Stopping interrupts penalizes all processes when one process executes a critical section
  - Stops all I/O activity
  - Restricts execution to one process for the entire system

- Can interfere with the scheduling invariant (low-priority process can block a high-priority process for which I/O has completed)

- Does not permit a data access policy
High-Level Mutual Exclusion

• Idea is to create a facility with the following properties
  – Permit designer to specify multiple critical sections
  – Allow independent control of each critical section
  – Provide an access policy (e.g., FIFO)

• A single mechanism, the *counting semaphore*, suffices
Counting Semaphore

- Operating system abstraction
- Instance can be created dynamically
- Each instance given unique name
  - Typically an integer
  - Known as *semaphore id*
- Instance consists of a tuple (count, set)
  - *Count* is an integer
  - *Set* is a set of processes waiting on the semaphore
Operations On Semaphores

- *Create* new semaphore
- *Delete* existing semaphore
- *Wait* on existing semaphore
  - Decrements count
  - Adds calling process to set waiting if resulting count is negative
- *Signal* existing semaphore
  - Increments count
  - Makes a process ready if any waiting
Semaphore Invariant

- Establishes relationship between conceptual purpose and implementation
- Must be re-established after each operation
- Surprisingly elegant:

A nonnegative semaphore count means that the set is empty. A count of negative \( N \) means that the set contains \( N \) waiting processes.
Counting Semaphores In Xinu

• Stored in an array of semaphore entries

• Each entry
  – Corresponds to one instance
  – Contains an integer count and pointer to list of processes

• Semaphore ID is index into array

• Policy for management of waiting processes is FIFO

• Each process that is enqueued on a semaphore queue is in the WAITING state
State Transitions With Waiting State

- **WAITING**
  - Transition to READY
  - Transition to CURRENT
  - Transition to SUSPENDED

- **READY**
  - Transition to CURRENT
  - Transition to SUSPENDED

- **CURRENT**
  - Transition to READY
  - Transition to SUSPENDED

- **SUSPENDED**
  - Transition to READY
  - Transition to CURRENT

Transitions:
- **signal**
- **wait**
- **resched**
- **suspend**
- **resume**
- **create**
Semaphore Definitions

/* semaphore.h - isbadsem */

#ifdef NSEM
#define NSEM 45    /* 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 queue;    /* queue of processes that are waiting */
        /* on the semaphore */
};

extern struct sentry semtab[];

#define isbadsem(s)   ((int32)(s) < 0 || (s) >= NSEM)
Implementation Of Wait

/* excerpt from wait.c - wait */

/*-----------------------------------------------
 * 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 semaphore table entry */

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

    semptr = &semtab[sem];
    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->sqeueue); /* enqueue on semaphore */
        resched(); /* and reschedule */
    }
    restore(mask);
    return OK;
}
Uses Of Semaphores

- Mutual exclusion
- Direct synchronization (e.g., producer-consumer)
Cooperative Mutual Exclusion

- Initialization

\[
\text{sid} = \text{semcreate}(1);
\]

- Use: bracket critical sections of code with calls to \textit{wait} and \textit{signal}

\[
\text{wait}(\text{sid});
\]

\[
\ldots \text{critical section (use shared resource)} \ldots
\]

\[
\text{signal}(\text{sid});
\]
Producer-Consumer Synchronization

• Typical scenario
  – Shared circular buffer
  – Producing process deposits items into buffer
  – Consuming process extracts items from buffer

• Must guarantee
  – Producer blocks when buffer full
  – Consumer blocks when buffer empty

• Can use two semaphores for synchronization
Producer-Consumer Synchronization

- Initialization

  ```
  psem = semcreate(buffer-size);
  csem = semcreate(0);
  ```

- Use by producer

  ```
  repeat forever {
    wait(psem);
    fill_next_buffer_slot;
    signal(csem);
  }
  ```
Producer-Consumer Synchronization
(continued)

- Use by consumer

    repeat forever {
        wait(csem);
        extract_from_buffer_slot;
        signal(psem);
    }

Illustration Of Producer-Consumer

- \( csem \) counts items currently in buffer
- \( psem \) counts unused slots in buffer
Semaphore Queuing Policy

- Used when *signal* called
- Determines which process to select among those waiting
- Examples
  - First-Come-First-Served (FCFS or FIFO)
  - Process priority
  - Random
Question

• The goal is “fairness”
• Which semaphore queuing policy implements goal best?
• In other words, how should we interpret fairness?
  – Should a low-priority process be allowed to run if a high-priority process is also waiting?
  – Should a low-priority process be blocked forever if high-priority processes use a resource?
Choosing A Semaphore Queueing Policy

- Difficult
- No single best answer
  - Fairness not easy to define
  - Scheduling and coordination interact
  - May affect other OS policies
- Interactions of heuristics may produce unexpected results
Example Semaphore Queuing Policy

- First-come-first-serve
- Straightforward to implement
- Works well for traditional uses of semaphores
- Potential problem: low-priority process can access while high-priority process remains waiting
Implementation Of FIFO Semaphore Policy

- Each semaphore uses a list to manage waiting processes
- List is run as a queue: insertions at one end and deletions at the other
- Example implementation follows
/* signal.c – signal */

/*-----------------------------------------------
 * 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 entry *sempt; /* ptr to semaphore table entry */

    mask = disable();
    if (isbadsem(sem)) {
        restore(mask);
        return SYSERR;
    }
    sept = &semtab[sem];
    if (sept->sstate == S_FREE) {
        restore(mask);
        return SYSERR;
    }
    if ((sept->scount++) < 0) { /* release a waiting process */
        ready(dequeue(sept->sqeue), RESCHED_YES);
    }
    restore(mask);
    return OK;
}
Semaphore Allocation

- **Static**
  - Semaphores defined at compile time
  - More efficient, but less powerful

- **Dynamic**
  - Semaphore created at runtime
  - More flexible
Xinu Semcreate (part 1)

/* semcreate.c - semcreate, newsem */

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;
}
/ * newem - 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 = SUSED;
            return sem;
        }
    }
    return SYSERR;
}
Semaphore Deletion

- Processes may be waiting
- Must choose a disposition for each
- Example: make process ready
/* semdelete.c - semdelete */

#include <xinu.h>

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

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

    sempttr = &semtab[sem];
    if (sempttr->sstate == S_FREE) {
        restore(mask);
        return SYSERR;
    }
    sempttr->sstate = S_FREE;
Xinu Semdelete (part 2)

while (semptr->scount++ < 0) { /* free all waiting processes */
    ready(getfirst(semptr->squeue), RESCHED_NO);
}
resched();
restore(mask);
return OK;
}
Do you understand semaphores?
Thought Problem
(The Convoy)

- One process creates a semaphore

    \[
    \text{mutex} = \text{sc}reate(1);
    \]

- Three processes execute the following

    ```
    \text{PROCESS} \text{convoy}(\text{char}_\text{to}_\text{print})
    \text{do} \text{forever} \{ 
    \text{think} \ (\text{i.e., use CPU});
    \text{wait}(\text{mutex});
    \text{print}(\text{char}_\text{to}_\text{print});
    \text{signal}(\text{mutex});
    \}
    ```

- The processes print characters \( A \), \( B \), and \( C \), respectively
Convoy Problem
(continued)

- Initial output
  - 20 $A$’s, 20 $B$’s, 20 $C$’s, 20 $A$’s, etc.

- After tens of seconds
  $ABCABCABC...$

- Facts
  - Everything is correct
  - No other processes are executing
  - Print is nonblocking (polled I/O)
Convoy Problem
(continued)

- Questions
  - How long is thinking time?
  - Why does convoy start?
  - Will output switch back given enough time?
  - Did knowing the policies or the implementation of the scheduler and semaphore mechanisms make the convoy behavior obvious?
Summary

• Process synchronization fundamental
  – Supplied to applications
  – Used inside OS

• Low-level mutual exclusion
  – Masks hardware interrupts
  – Avoids rescheduling
  – Insufficient for all coordination
Summary (continued)

- High-level coordination
  - Used by subsets of processes
  - Available inside and outside OS
  - Implemented with counting semaphore

- Counting semaphore
  - Powerful abstraction
  - Provides mutual exclusion and producer/consumer synchronization