Chapter 2
Processes and Threads

[...]  
2.3 Interprocess communication  
2.4 Classical IPC problems  
2.5 Scheduling

Interprocess Communication  
Race Conditions

Two processes want to access shared memory at same time
Critical Regions (1)

Four conditions to provide mutual exclusion
1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Critical Regions (2)

Mutual exclusion using critical regions
Mutual Exclusion with Busy Waiting (1)

while (TRUE) {
    while (turn != 0) /* loop */;
    critical_region();
    turn = 1;
    noncritical_region();
}

(a)

while (TRUE) {
    while (turn != 1) /* loop */;
    critical_region();
    turn = 0;
    noncritical_region();
}

(b)

Proposed solution to critical region problem
(a) Process 0. (b) Process 1.

Mutual Exclusion with Busy Waiting (2)

#define FALSE 0
#define TRUE 1
#define N 2 /* number of processes */

int turn; /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other; /* number of the other process */
    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process; /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}

Peterson's solution for achieving mutual exclusion
Mutual Exclusion with Busy Waiting (3)

enter_region:
TSL REGISTER, LOCK | copy lock to register and set lock to 1
CMP REGISTER, #0 | was lock zero?
JNE enter_region | if it was non zero, lock was set, so loop
RET | return to caller; critical region entered

leave_region:
MOVE LOCK, #0 | store a 0 in lock
RET | return to caller

Entering and leaving a critical region using the TSL instruction

---

Sleep and Wakeup

```c
#define N 100
int count = 0; /* number of slots in the buffer */
/* number of items in the buffer */

void producer(void)
{
    int item;
    while (TRUE) {
        item = produce_item(); /* repeat forever */
        if (count == N) sleep(); /* generate next item */
        if (item) { /* if buffer is full, go to sleep */
            insert_item(item);
            count = count + 1;
        }
        if (count == 1) wakeup(consumer); /* increment count of items in buffer */
        if (count == 0) sleep(); /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;
    while (TRUE) {
        if (count == 0) sleep(); /* repeat forever */
        if (item) { /* if buffer is empty, go to sleep */
            item = remove_item(); /* take item out of buffer */
            count = count - 1;
        }
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item); /* print item */
    }
}
```

Producer-consumer problem with fatal race condition
Semaphores

- Generalize Sleep and Wakeup, counting
  - Checking and updating atomically
- Down: if value > 0, decrement and proceed
  - If value = 0, suspend (sleep)
- Up: increment value
  - If one or more procs suspended, awake

```c
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;
    while (TRUE)
    {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;
    while (TRUE)
    {
        down(&mutex);
        down(&full);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

The producer-consumer problem using semaphores
Mutexes

mutex_lock:
TSL REGISTER,MUTEX | copy mutex to register and set mutex to 1
CMP REGISTER,#0 | was mutex zero?
JZE ok | if it was zero, mutex was unlocked, so return
CALL thread_yield | mutex is busy; schedule another thread
JMP mutex_lock | try again later
ok: RET | return to caller; critical region entered

mutex_unlock:
MOVE MUTEX,#0 | store a 0 in mutex
RET | return to caller

Implementation of \textit{mutex\_lock} and \textit{mutex\_unlock}

Mutex: Software and Hardware

busy waiting
– incorrect solutions, understanding concurrency
– software: Peterson’s protocol
– hardware: via TSL and protocols

without busy-waiting
– sleep and wakeup, producer/consumer
– semaphores
– mutexes
– monitors
– message passing, barriers
Monitors (1)

```plaintext
monitor example
  integer i;
  condition c;

  procedure producer();
  .
  .
  end;

  procedure consumer();
  .
  .
  end;
end monitor:

Example of a monitor
```

Monitors (2)

```plaintext
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
end monitor:

procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
end;

• Outline of producer-consumer problem with monitors
  – only one monitor procedure active at one time
  – buffer has $N$ slots
```
Monitors (3)

• Prologue Code executed when starting execution of a monitor's operation:
  \[ P(mutex); \]
• Epilogue Code executed when ending execution of a monitor's operation:
  \[ \text{if next\_count > 0 then } V(next) \text{ else } V(mutex); \]

Monitors (4)

type condition is record
  count : integer initialized to 0;
  queue : semaphore initialized to 0; end;
procedure wait (x : in out condition) is
  begin
    x.count := x.count + 1;
    if next_count > 0 then V(x.queue) else V(mutex);
    P(x.queue);
    x.count := x.count - 1;
  end;
procedure signal (x : in out condition) is
  begin
    if x.count > 0 then begin
      next_count := next_count + 1;
      V(x.queue); P(next);
      next_count := next_count - 1;
    end;
end;

Hoare Monitor implementation using semaphores
Message Passing

```c
#define N 100

void producer(void) {
    int item;
    message m; /* message buffer */
    while (TRUE) {
        item = produce_item(); /* generate something to put in buffer */
        receive(consumer, &m); /* wait for an empty to arrive */
        build_message(&m, item);
        send(consumer, &m); /* construct a message to send */
        /* send item to consumer */
    }
}

void consumer(void) {
    int item, i;
    message m;
    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m); /* get message containing item */
        item = extract_item(&m); /* extract item from message */
        if (item == i) { /* send back empty reply */
            send_back_empty_reply();
        }
        consume_item(item); /* do something with the item */
    }
}
```

The producer-consumer problem with N messages

Barriers

- **Use of a barrier**
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through
More about concurrency

- Dining Philosophers
- Sleeping Barber
- Scheduling
  - introduction & goals
  - batch systems: shortest-job-first
  - round-robin, interactive, real-time
  - policy versus mechanism (!)
  - thread scheduling

Dining Philosophers (1)

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock
Dining Philosophers (2)

#define N 5            /* number of philosophers */

void philosopher(int i) /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();       /* philosopher is thinking */
        take_fork(i);  /* take left fork */
        take_fork((i+1) % N); /* take right fork; % is modulo operator */
        eat();         /* yum-yum, spaghetti */
        put_fork(i);   /* put left fork back on the table */
        put_fork((i+1) % N); /* put right fork back on the table */
    }
}

A nonsolution to the dining philosophers problem

Dining Philosophers (3)

#define N  5            /* number of philosophers */
#define LEFT  (i+N−1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N   /* number of i's right neighbor */
#define THINKING 0       /* philosopher is thinking */
#define HUNGRY 1         /* philosopher is trying to get forks */
#define EATING 2         /* philosopher is eating */

typedef int semaphore;  /* semaphores are a special kind of int */
int state[N];           /* array to keep track of everyone's state */
semaphore mutex = 1;    /* mutual exclusion for critical regions */
semaphore s[N];         /* one semaphore per philosopher */

void philosopher(int i) /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {
        think();       /* repeat forever */
        take_forks(i); /* philosopher is thinking */
        eat();         /* yum-yum, spaghetti */
        put_forks(i);  /* put both forks back on table */
    }
}

Solution to dining philosophers problem (part 1)
Dining Philosophers (4)

```c
void take_forks(int i) {
    down(&mutex);    /* enter critical region */
    state[i] = HUNGRY;
    test(i);         /* record fact that philosopher i is hungry */
    up(&mutex);      /* try to acquire 2 forks */
    down(&mutex);    /* block if forks were not acquired */
}

void put_forks(i) {
    down(&mutex);    /* enter critical region */
    state[i] = THINKING;
    test(LEFT);      /* philosopher has finished eating */
    test(RIGHT);     /* see if left neighbor can now eat */
    up(&mutex);      /* see if right neighbor can now eat */
    down(&mutex);    /* exit critical region */
}

void test(i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&mutex);
    }
}
```

Solution to dining philosophers problem (part 2)

The Readers and Writers Problem

```c
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void) {
    while (TRUE) {
        down(&mutex);    /* use your imagination */
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void) {
    while (TRUE) {
        think_up_data();    /* repeat forever */
        down(&db);    /* get exclusive access to 'rc' */
        think_up_data();    /* noncritical region */
        write_data_base();    /* update the data */
        up(&db);    /* release exclusive access */
    }
}
```

A solution to the readers and writers problem
The Sleeping Barber Problem (1)

Solution to sleeping barber problem.
Introduction to Scheduling (1)

- Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound process
  - an I/O bound process

Introduction to Scheduling (2)

When to Schedule
- process creation, fork: parent or child?
- process exit, choose successor, if none, sys proc
- process blocks: I/O, semaphore
  - perhaps reason for blocking helps choose (critical region)
  - usually scheduler is unaware! Why? Scaling
- I/O interrupt
- H/W clock interrupts define quanta
  - Preemptive versus Non-preemptive scheduling
Introduction to Scheduling (3)

All systems
Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems
Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems
Response time - respond to requests quickly
Proportionality - meet users’ expectations

Real-time systems
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Scheduling Algorithm Goals

Scheduling in Batch Systems (1)

• FCFS (almost like FIFO)
• Shortest Job First
• Shortest Remaining Time Next
• Three-level Scheduling (not just CPU)
Scheduling in Batch Systems (2)

An example of shortest job first scheduling

Scheduling in Batch Systems (2)

Three level scheduling
Scheduling in Interactive Systems (1)

- Round Robin Scheduling
  - list of runnable processes
  - list of runnable processes after B uses up its quantum
- Quantum length – how to choose?
- Context/process switch -> fragmentation (OS theme)

Scheduling in Interactive Systems (2)

Scheduling algorithm with four priority classes
- advantages: finer QoS / cautions: starvation
- dynamic priorities, aging; UNIX: “nice”
- within priority class, typically round-robin
Interactive Scheduling Schemes

- Shortest Process Next
- Guaranteed Scheduling
  - $1/n$ of CPU if $n$ users present (user v. process)
  - compute ratio used to ratio entitled, adjust
- Lottery Scheduling
- Fair-Share – explicit user v. process
- in the limit: Real-Time

Scheduling in Real-Time Systems

Schedulable real-time system (oversimplified)

- Given
  - $m$ periodic events
  - event $i$ occurs within period $P_i$ and requires $C_i$ seconds
- Then the load can only be handled if
Scheduling classified

- **when** to schedule = decision mode
  - preemptive (cheaper), nonpreemptive (more general)
- **who** to schedule = priority function + arbitration rule

At points specified by the decision mode, scheduler evaluates priority function for all; if there are ties, uses arbitration rule

Scheduling classified

priority function parameters

- attained service time
  - time using CPU since arrival; common measure
- real-time in system
  - attained + waiting time (ready but not running)
- total service time
  - will consume in its lifetime, sometimes predictable
- deadline; periodicity
  - when it must be completed, how often run
- external priority – e.g., business policies
- memory requirements (important in batch; also swapping overhead for interactive)
Policy versus Mechanism

• Separate what is **allowed** to be done with **how** it is done
  – a process knows which of its children threads are important and need priority

• Scheduling algorithm parameterized
  – mechanism in the kernel

• Parameters filled in by user processes
  – policy set by user process

Thread Scheduling (1)

Possible scheduling of user-level threads
• 50-msec process quantum
• threads run 5 msec/CPU burst
Thread Scheduling (2)

Possible scheduling of kernel-level threads
• 50-msec process quantum
• threads run 5 msec/CPU burst

Thread Scheduling (3)

• user-threads switch (i.e., within same process) is inexpensive
• kernel-threads switch across process is full process context switch
• user-level threads implement their own scheduler – knowing application
• other models (exokernels, etc.) exist