Operating Systems

Process Synchronization
(Ch 2.3, 2.4)

Cooperating Processes

• Consider: print spooler
  – Enter file name in spooler queue
  – Printer daemon checks queue and prints

  • “Race conditions” (ugh!)
  • (Hey, you! Show demo!)

Outline

• Need for synchronization
  – why?
• Solutions that require busy waiting
  – what?
• Semaphores
  – what are they?
• Classical problems
  – dining philosophers
  – readers/writers

Producer Consumer

• Model for cooperating processes
• Producer “produces” and item that consumer “consumes”
• Bounded buffer (shared memory)
  item buffer[MAX]; /* queue */
  int counter; /* num items */

Producer

item i; /* item produced */
int in; /* put next item */
while (1) {
  produce an item
  while (counter == MAX){/*no-op*/}
  buffer[in] = item;
  in = (in + 1) % MAX;
  counter = counter + 1;
}
Consumer
item i; /* item consumed */
int out; /* take next item */
while (1) {
    while (counter == 0) {/*no-op*/}
    item = buffer[out];
    out = (out + 1) % MAX;
    counter = counter - 1;
    /*consume the item*/
}

Critical Section
- Mutual Exclusion
  - Only one process inside critical region
- Progress
  - No process outside critical region may block other
    processes wanting in
- Bounded Waiting
  - No process should have to wait forever (starvation)
- Note, no assumptions about speed!

First Try: Strict Alternation
int turn; /* shared, id of turn */
while(1) {
    while (turn <> my_pid) { /* no-op */}
    /* critical section */
    turn = your_pid
    /* remainder section */
}

Second Try
int flag[1]; /* boolean */
while(1) {
    flag[my_pid] = true;
    while (flag[your_pid]) { /* no-op */}
    /* critical section */
    flag[my_pid] = false;
    /* remainder section */
}

Third Try: Peterson’s Solution
int flag[1]; /* boolean */
int turn;
while(1) {
    flag[my_pid] = true;
    turn = your_pid;
    while (flag[your_pid] && turn==your_pid){ /* no-op */}
    /* critical section */
    flag[my_pid] = false;
    /* remainder section */
}

Trouble!
P: R1 = counter (R1 = 5)
P: R1 = R1 + 1 (R1 = 6)
C: R2 = counter (R2 = 5)
C: R2 = R2 -1 (R2 = 4)
C: counter = R2 (counter = 4)
P: counter = R1 (counter = 6)
Multiple-Processes

- “Bakery Algorithm”
- Common data structures
  - boolean choosing[n];
  - int num[n];
- Ordering of processes
  - If same number, can decide “winner”

```c
choosing[my_pid] = true;
num[my_pid] = max(num[0],num[1] ...)+1
choosing[my_pid] = false;
for (j=0; j<n; j++) {
  while(choosing[j]) { }
  while(num[j]!=0 &&
    (num[j],j)<(num[my_pid],my_pid)){} }
/* critical section */
num[my_pid] = 0;
```

Synchronization Hardware

- Test-and-Set: returns and modifies atomically

```c
int Test_and_Set(int &target) {
  int temp;
  temp = target;
  target = true;
  return temp;
}
```

Using Test_and_Set

```c
while(1) {
  while (Test_and_Set(lock)) { }
  /* critical section */
  lock = false;
  /* remainder section */
}
```

All the solutions so far have required “Busy Waiting” … what is that?

Outline

- Need for synchronization (done)
  - why?
- Solutions that require busy waiting (done)
  - what?
- Semaphores
  - what are they?
- Classical problems
  - dining philosophers
  - reader/writers

Semaphores

- Do not require “busy waiting”
- Semaphore S (shared, often initially =1)
  - integer variable
  - accessed via two (indivisible) atomic operations
  - wait(S): S = S - 1
    if S<0 then block(S)
  - signal(S): S = S + 1
    if S<=0 then wakeup(S)
**Critical Section w/Semaphores**

```c
semaphore mutex; /* shared */
while(1) {
    wait(mutex);
    /* critical section */
    signal(mutex);
    /* remainder section */
}
```

(Hey, you! Show demo!)

**Semaphore Implementation**

- Disable interrupts
  - Why is this not evil?
  - Multi-processors?
- Use correct software solution
- Use special hardware, i.e.- Test-and-Set

**Design Technique: Reducing a Problem to a Special Case**

- Simple solution not adequate
  - ex: disabling interrupts
- Problem solution requires special case solution
  - ex: protecting $S$ for semaphores
- Simple solution adequate for special case
- Other examples:
  - name servers, on-line help

**Trouble!**

```
signal(S)  /* cr */
wait(S)
wait(S)  /* cr */
wait(S)
```

Process A

```
wait(S)
wait(Q)
...```

Process B

```
wait(Q)
wait(S)
...```

**Classical Synchronization Problems**

- Bounded Buffer
- Readers Writers
- Dining Philosophers

**Dining Philosophers**

- Philosophers
  - Think
  - Sit
  - Eat
  - Think
- Need 2 chopsticks to eat
Dining Philosophers

```
Philosopher i:
while (1) {
    /* think... */
    wait(chopstick[i]);
    wait(chopstick[i+1 % 5]);
    /* eat */
    signal(chopstick[i]);
    signal(chopstick[i+1 % 5]);
}
(Other solutions?)
```

Other Solutions

- Allow at most N-1 to sit at a time
- Allow to pick up chopsticks only if both are available
- Asymmetric solution (odd L-R, even R-L)

Readers-Writers

- **Readers** only read the content of object
- **Writers** read and write the object
- Critical region:
  - No processes
  - One or more readers (no writers)
  - One writer (nothing else)
- Solutions favor Reader or Writer

```
Readers-Writers

Shared:
semaphore mutex, wrt;
int readcount;

Writer:
wait(wrt)
/* write stuff */
signal(wrt);
```

Readers-Writers

```
Readers-Writers

Reader:
wait(mutex);
readcount = readcount + 1;
if (readcount==1) wait(wrt);
signal(mutex);
/* read stuff */
wait(mutex);
readcount = readcount - 1;
if (readcount==0) signal(wrt);
signal(mutex);
```

Monitors

- High-level construct
- Collection of:
  - variables
  - data structures
  - functions
  - Like C++ class
- One process active inside
- “Condition” variable
  - not counters like semaphores
Monitor Producer-Consumer

```c
monitor ProducerConsumer {
  condition full, empty;
  integer count;
  /* function prototypes */
  void enter(item i);
  item remove();
}
void producer()
{
  item i;
  while (1) {
    /* produce item i */
    ProducerConsumer.enter(i);
  }
}
void consumer()
{
  item i;
  while (1) {
    i = ProducerConsumer.remove();
    /* consume item i */
  }
}
```

Monitor Producer-Consumer

```c
void enter (item i) {
  if (count == N) sleep(full);
  /* add item i */
  count = count + 1;
  if (count == 1) then wakeup(empty);
}
item remove () {
  if (count == 0) then wakeup(empty);
  /* remove item into i */
  count = count - 1;
  if (count == N-1) then sleep(full);
  return i;
}
```

Other Process Synchronization Methods

- Sequencers
- Path Expressions
- Serializers
- ...

All essentially equivalent in terms of semantics. Can build each other!

Trouble?

- Monitors and Regions attractive, but ...
  - Not supported by C, C++, Pascal ...  
    - semaphores easy to add
- Monitors, Semaphores, Regions ...
  - require shared memory
  - break on multiple CPU (w/own mem)
  - break distributed systems
- In general, Inter-Process Communication (IPC)
  - Move towards Message Passing

Inter Process Communication

- How does one process communicate with another process? Some of the ways:
  - shared memory – read/write to shared region
    - shmget(), shmat(), shmdt() in Unix
  - Memory mapped files in WinNT/2000
  - semaphores - signal notifies waiting process
  - software interrupts - process notified asynchronously
  - pipes - unidirectional stream communication
  - message passing - processes send and receive messages.
Software Interrupts

- Similar to hardware interrupt.
- Processes interrupt each other (often for system call).
- Asynchronous! Stops execution then restarts
  - `cntl-C`
  - child process completes
  - alarm scheduled by the process expires
    - Unix: `SIGNALRM` from `alarm()` or `setitimer()`
    - resource limit exceeded (disk quota, CPU time...)
    - programming errors: invalid data, divide by zero

Send Interrupt

- `SendInterrupt(pid, num)`
  - type num to process `pid`
- `kill()` in Unix
  - (NT doesn’t allow signals to processes)

Handle Interrupt

- `HandleInterrupt(num, handler)`
  - type num, use function handler
  - `signal()` in Unix
  - Use exception handler in WinNT/2000

- Typical handlers:
  - ignore
  - terminate (maybe w/core dump)
  - user-defined

(Hey, show demos!)

Unreliable Signals

- Before POSIX.1 standard:
  - `signal(SIGINT, sig_int);`
  - `sig_int()` {
    - `/* re-establish handler */`
    - `signal(SIGINT, sig_int);`
  }
- Another signal could come before handler re-established!

Pipes

- One process writes, 2nd process reads
  - `% ls | more`
  - Shell:
    - 1 create a pipe
    - 2 create a process for `ls` command, setting `stdout` to write side of pipe
    - 3 create a process for `more` command, setting `stdin` to read side of pipe
  - Shell:
    - `1 create a pipe`
    - `2 create a process for ls command, setting stdout to write side of pipe`
    - `3 create a process for more command, setting stdin to read side of pipe`

The Pipe

- Process inherits file descriptors from parent
  - file descriptor 0 stdin, 1 stdout, 2 stderr
- Process doesn’t know (or care!) when reading from keyboard, file, or process or writing to terminal, file, or process
- System calls:
  - `read(fd, buffer, nbytes)` (`scanf()` built on top)
  - `write(fd, buffer, nbytes)` (`printf()` built on top)
  - `pipe()` creates a pipe
    - rgfd array of 2 fd. Read from rgfd[0], write to rgfd[1]
- (Hey, show sample code!)

Bounded Buffer

- shared buffer (Unix 4096K)
- block writes to full pipe
- block reads to empty pipe

(Hey, show demos!)
Message Passing

- Communicate information from one process to another via primitives:
  - send(dest, &message)
  - receive(source, &message)
- Receiver can specify ANY
- Receiver can block (or not)

Producer-Consumer

```c
void Producer() {
    while (TRUE) {
        /* produce item */
        build_message(&m, item);
        send(consumer, &m); /* wait for ack */
    }
}
void Consumer() {
    while(1) {
        receive(producer, &m);
        extract_item(&m, &item);
        send(producer, &m); /* ack */
        /* consume item */
    }
}
```

Consumer Mailbox

```c
void Consumer() {
    for (i=0; i<N; i++)
        send(producer, &m); /* N empties */
    while(1) {
        receive(producer, &m);
        extract_item(&m, &item);
        send(producer, &m); /* ack */
        /* consume item */
    }
}
```

New Troubles with Messages?

- Scrambled messages (checksum)
- Lost messages (acknowledgements)
- Lost acknowledgements (sequence no.)
- Process unreachable (down, terminates)
- Naming
- Authentication
- Performance (from copying, message building)
- (Take cs4513!)