Before We Begin ...

Read Chapter 7
- Semaphores
- Monitors
- Message-Passing

Programming Assignment #1 (see CSE120 web page)
- Due in October 19
- Contact a TA if you need an account
- Start early!

Cooperating Processes

Why structure a computation as cooperative processes?

Performance: speed
- Exploit inherent parallelism of computation
- Allow some parts to proceed while others do I/O

Modularity: reusable self-contained programs
- Each may do a useful task on its own
- May also be useful as a sub-task for others

Examples of Cooperating Processes

Pipeline
- \( P_1 \rightarrow P_2 \rightarrow P_3 \)

Client/Server
- \( C \)

Parent/Child
- \( P \)
- \( C_1, C_2, C_3 \)
Interprocess Communication

In order to cooperate, need to be able to communicate

Achieved via IPC: interprocess communication
  • ability for a process to communicate with another

Interprocess communication requires
  • information transfer
  • synchronization

Need mechanisms for each

The Producer/Consumer Problem

Producer process produces data
Consumer process consumes data

Cooperation: data from Producer is fed to Consumer
  • How does data get from Producer to Consumer?
  • How does Consumer wait for Producer?

Producer/Consumer: Shared Memory

shared int buffer[N]; /* shared memory */

Producer
  int nextin = 0;
  while (TRUE) {
    buffer[nextin%N] = produce ();
    nextin++;
  }

Consumer
  int nextout = 0;
  while (TRUE) {
    consume (buffer[nextout%N]);
    nextout++;
  }

Problems
  • No synchronization (only shared memory)
  • No mutual exclusion for critical sections

Add Semaphore for Mutual Exclusion

shared int buffer[N];
shared sem mutex = 1;

Producer
  int nextin = 0;
  while (TRUE) {
    Wait (mutex);
    buffer[nextin%N] = produce ();
    Signal (mutex);
    nextin++;
  }

Consumer
  int nextout = 0;
  while (TRUE) {
    Wait (mutex);
    consume (buffer[nextout%N]);
    Signal (mutex);
    nextout++;
  }

Problem: still no synchronization
Add Semaphores for Synchronization

Producer
int nextin = 0;
while (TRUE) {
    Wait (emptyslots);
    Wait (mutex);
    buffer[nextin%N] = produce();
    Signal (mutex);
    nextin++;
}

Consumer
int nextout = 0;
while (TRUE) {
    Wait (fullslots);
    Wait (mutex);
    consume(buffer[nextout%N]);
    Signal (mutex);
    nextout++;
}

Works, but not easy to understand: easily leads to bugs

Monitors

Programming language construct for IPC

- Variables requiring controlled access (shared mem)
- Accessed via procedures (mutual exclusion)
- Condition variables (synchronization)
  - Wait (cond), Signal (cond)

Only one process can be active inside the monitor

- “active” means running or able to run
- others must wait

Producer/Consumer Using a Monitor

```
monitor ProducerConsumer {
    condition slotavail, itemavail;
    int count = 0, nextin = 0, nextout = 0, buffer[N];

    PutItem (item) int item; {
        if (count == N) Wait (slotavail);
        buffer[nextin%N] = item;
        nextin++;
        count++;
        if (count == 1) Signal (itemavail);
    }

    GetItem (item) int item; { 
        if (count == 0) Wait (itemavail);
        item = buffer[nextout%N];
        nextout++
        count--;
        if (count == N-1) Signal (slotavail);
        return (item);
    }

    Producer while (TRUE) {
        PutItem (produce());
    }

    Consumer while (TRUE) {
        consume (GetItem());
    }
}
```
Issues with Monitors

Given $P_1$ waiting on condition $c$, and $P_2$ signals $c$
- now, $P_1$ and $P_2$ able to run (breaks mutual exclusion)
- one solution: signal statements must be at end of monitor procedures

Condition variables have no memory
- a signal without someone waiting does nothing
- signal is “lost” (no memory, no future effect)

Monitors bring structure to IPC
- localizes critical sections and synchronization

Message Passing

Operating system mechanism for IPC
- Send (destination, message)
- Receive (source, message)

Data transfer: into and out of kernel message buffers
Synchronization: can't receive until message is sent

Producer/Consumer: Message-Passing

With Flow Control

```c
/* NO SHARED MEMORY */

Producer
    int item;
    message msg;

while (TRUE) {
    item = produce ();
    insert (&msg, item);
    Send (Consumer, &msg);
}

Consumer
    int item;
    message msg, empty;

do N times {
    Send (Producer, &empty);
}

while (TRUE) {
    Receive (Consumer, &empty);
    item = extract (&msg);
    consume (item);
}
```
An Optimization

**Producer**

```c
int item;
message msg;

while (TRUE) {
    item = produce ();
    Receive (Consumer, &msg);
    insert (&msg, item);
    Send (Consumer, &msg);
}
```

**Consumer**

```c
int item;
message msg;

do N times {
    Send (Producer, &msg);
}

while (TRUE) {
    Receive (Producer, &msg);
    item = extract (&msg);
    Send (Producer, &msg);
    consume (item);
}
```

Issues with Message Passing

Who should messages be addressed to?
- processes or ports (mailboxes)

What if a process wants to receive from anyone?
- `pid = receive (*, msg)`

Synchronous (blocking) vs. asynchronous (non-blocking)
- typically, send is non-blocking, receive is blocking

Kernel buffering: how many sends without receives?
Good paradigm for IPC over networks (no shared mem)