Before We Begin ...

Read Chapter 6 (Process Synchronization)
  • Semaphores, Monitors, Message Passing

Programming Assignment #2
  • Due Sunday April 23
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**

\[ C \rightarrow \quad \quad \quad \rightarrow S \]

**Parent/Child**

\[ P \rightarrow C_1 \quad C_2 \quad C_3 \]
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
  • data transfer
  • synchronization

Need mechanisms for each
The Producer/Consumer Problem

Producer produces data and places in shared buffer

Consumer consumes data removed from buffer

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 buf[N], in = 0, out = 0;

**Producer**
while (TRUE) {
    buf[in] = Produce ();
    in = (in + 1)%N;
}

**Consumer**
while (TRUE) {
    Consume (buf[out]);
    out = (out + 1)%N;
}

No synchronization
- Consumer must wait for something to be produced
- What about Producer?

No mutual exclusion for critical sections
- Why? There may be multiple producers/consumers
Add Semaphores for Synchronization

shared int buf[N], in = 0, out = 0;
shared sem filledslots = 0, emptyslots = N;

Producer
while (TRUE) {
    wait (emptyslots);
    buf[in] = Produce ();
    in = (in + 1) % N;
    signal (filledslots);
}

Consumer
while (TRUE) {
    wait (filledslots);
    Consume (buf[out]);
    out = (out + 1) % N;
    signal (emptyslots);
}

Buffer empty, Consumer waits
Buffer full, Producer waits
But, don’t confuse synchronization with mutual exclusion
Multiple Producers

Shared int buf[N], in = 0, out = 0;
Shared sem filledslots = 0, emptyslots = N;

Producer1
while (TRUE) {
    wait (emptyslots);
    buf[in] = Produce ();
    in = (in + 1)%N;
    signal (filledslots);
}

Producer2
while (TRUE) {
    wait (emptyslots);
    buf[in] = Produce ();
    in = (in + 1)%N;
    signal (filledslots);
}

Consumer
while (TRUE) {
    wait (filledslots);
    Consume (buf[out]);
    out = (out + 1)%N;
    signal (emptyslots);
}

There is now a race condition in Producers’ codes
- Inconsistent updating of variables buf and in

Need mutual exclusion
Add Semaphore for Mutual Exclusion

shared int buf[N], in = 0, out = 0;
shared sem filledslots = 0, emptyslots = N, mutex = 1;

Producer1, 2, ...
while (TRUE) {
    wait (emptyslots);
    wait (mutex);
    buf[in] = Produce ();
    in = (in + 1)%N;
    signal (mutex);
    signal (filledslots);
}

Consumer1, 2, ...
while (TRUE) {
    wait (filledslots);
    wait (mutex);
    Consume (buf[out]);
    out = (out + 1)%N;
    signal (mutex);
    signal (emptyslots);
}

Works for multiple producers and multiple consumers

But not easy to understand: easily leads to bugs

• What if wait statements were interchanged?
• Would two mutex semaphores (p, c) solve this?
Monitors

Programming language construct for IPC

- Variables requiring controlled access (shared mem)
- Accessed via procedures (mutual exclusion)
- Condition variables (synchronization)
  - wait (cond): block until another process signals cond
  - signal (cond): unblock a process waiting on cond

Only one process can be active inside the monitor

- “Active” = running or able to run; others must wait
An Analogy for Monitors

Gate enforces mutual exclusion:
- open if no process active in monitor
- closes when process enters
- opens when process exits or waits

- Waiting Area: Multiple processes can be waiting
- Active Area: Only one process can be active

Wait (cond): causes calling process to enter waiting area and gate to re-open
Signal (cond): causes a waiting process to re-enter active area; signaling process must exit immediately!
Producer/Consumer Using a Monitor

monitor ProducerConsumer {
    int buf[N], in = 0, out = 0, count = 0;
    cond slotavail, itemavail;

    PutItem (int item)
    {
        if (count == N) wait (slotavail);
        buf[in] = item;
        in = (in + 1)%N;
        count++;
        signal (itemavail);
    }

    GetItem ()
    {
        int item;
        if (count == 0) wait (itemavail);
        item = buf[out];
        out = (out + 1)%N;
        count--;
        signal (slotavail);
        return (item);
    }

    Consumer
    while (TRUE) {
        Consume (GetItem ());
    }

    Producer
    while (TRUE) {
        PutItem (Produce ());
    }
}
Issues with Monitors

Given P1 waiting on condition c, and P2 signals c

• P1 and P2 able to run: breaks mutual exclusion
• One solution: Signal just before returning

Condition variables have no memory

• 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_buffer)
- receive (source, message_buffer)

Data transfer: in to and out of kernel message buffers

Synchronization: can’t receive until message is sent
Producer/Consumer: Message-Passing

/* NO SHARED MEMORY */

Producer
int item;

while (TRUE) {
    item = Produce ();
    send (Consumer, &item);
}

Consumer
int item;

while (TRUE) {
    receive (Producer, &item);
    Consume (item);
}
With Flow Control

**Producer**

```c
int item, dummy;

while (TRUE) {
    receive (Consumer, &dummy);
    item = Produce ();
    send (Consumer, &item);
}
```

**Consumer**

```c
int item, dummy;

do N times {
    send (Producer, &dummy);
}

while (TRUE) {
    receive (Producer, &item);
    Consume (item);
    send (Producer, &dummy);
}
An Optimization

**Producer**

```c
int item, dummy;

while (TRUE) {
    item = Produce();
    receive (Consumer, &dummy);
    send (Consumer, &item);
}
```

**Consumer**

```c
int item, dummy;

do N times {
    send (Producer, &dummy);
}

while (TRUE) {
    receive (Producer, &item);
    send (Producer, &dummy);
    Consume (item);
}
```
Issues with Message Passing

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

What if a process wants to receive from anyone?
- \texttt{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)