CSE 120: Principles of Operating Systems
Lecture 6: InterProcess Communication (IPC)

Prof. Joseph Pasquale
University of California, San Diego

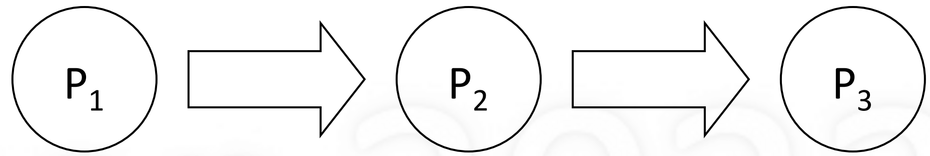
February 8, 2023 (updated 3/11/23)

Cooperating Processes

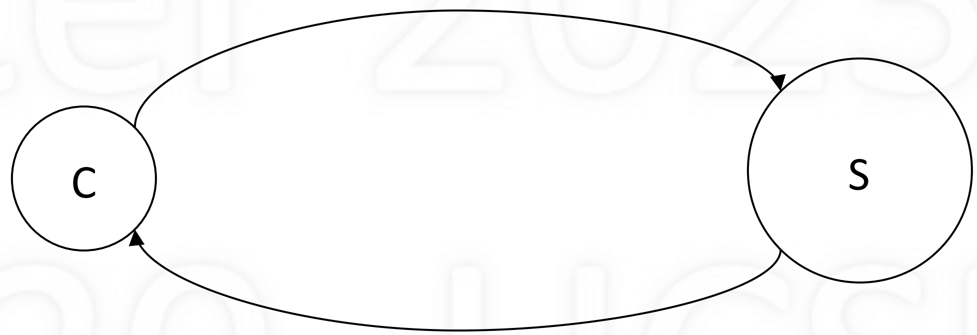
- Why structure a computation as a set of cooperating (communicating) processes?
- Performance: speed
  - Exploit inherent parallelism of computation
  - Allow some parts to proceed why others do I/O
- Modularity: reusable self-contained programs
  - Each may do a useful task on its own
  - May also be useful as a sub-task for others

Examples of Cooperating Processes

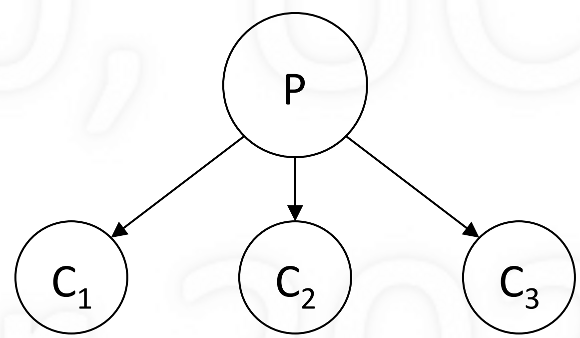
Pipeline



Client/Server



Parent/Child



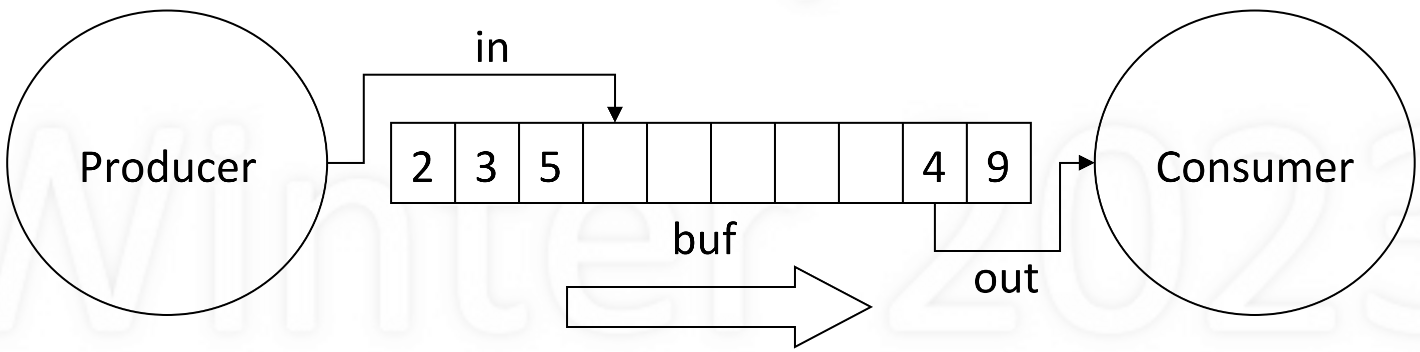
Inter-Process Communication

- To cooperate, need ability to communicate
- IPC: inter-process communication
  - Communication between processes
- IPC requires
  - *data transfer*
  - *synchronization*
- Need mechanisms for both

Three Abstractions for IPC

- Shared memory + semaphores
- Monitors
- Message passing

The Producer/Consumer Problem



- Producer produces data, inserts in shared buffer
- Consumer removes data from buffer, consumes it
- Cooperation: Producer feeds 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
  - Relevant if multiple producers or multiple consumers

Recall Semaphores

- Semaphore: synchronization variable
  - Takes on integer values
  - Has an associated list of waiting processes
- Operations

  wait (s)   { s = s–1; block if s < 0 }

  signal (s)   { s = s+1; unblock a process if any }
- No other operations allowed (e.g., can't test s)

Semaphores for Synchronization

```
shared int buf[N], in = 0, out = 0;
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
- General synchronization vs. mutual exclusion

Multiple Producers

```
shared int buf[N], in = 0, out = 0;
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 a race condition in the Producer code
- Inconsistent updating of variables buf and in
- Need mutual exclusion

Semaphore for Mutual Exclusion

```
shared int buf[N], in = 0, out = 0;
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 consumers
- But not easy to understand: can lead to bugs
  - example: what if wait statements interchanged?

Allowing For More Parallelism

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

Producer1, 2, …

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

Consumer1, 2, …

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

- Separate producer and consumer mutexes
- Decreases consumer/producer dependencies
  - increases parallelism

Monitors

- Programming language construct for IPC
  - Variables (shared) requiring controlled access
  - Accessed via procedures (mutual exclusion)
  - Condition variables (general synchronization)
    - wait (cond): block until another process signals cond
    - signal (cond): unblock a process waiting on cond
- Only one process can be active inside monitor
  - Active = running or able to run; others must wait

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);
                                            }
}
```

Producer

```
while (TRUE) {
   PutItem (Produce ());
}
```

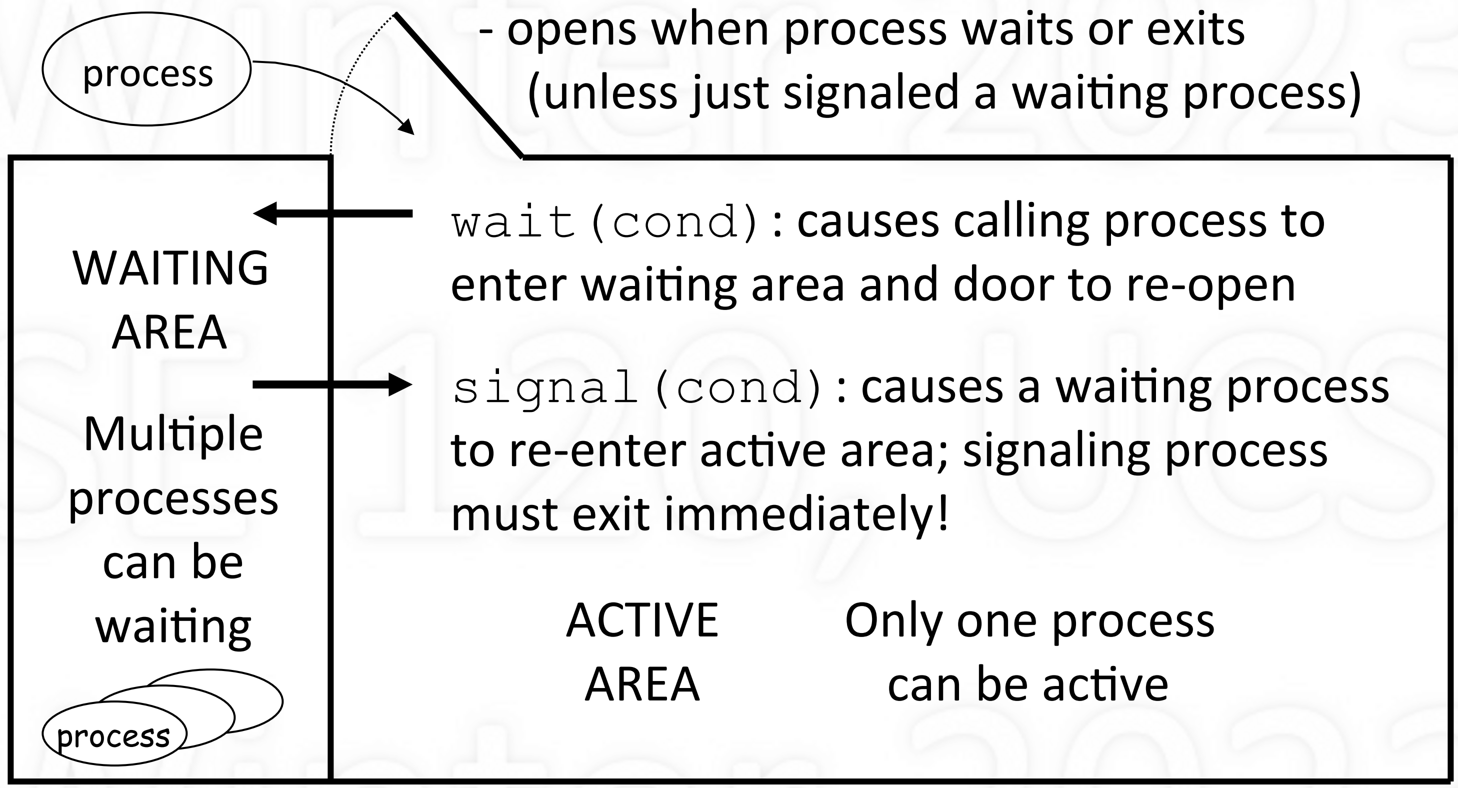
Consumer

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

How Synchronization Works

Door with “monitor lock” enforces mutual exclusion:

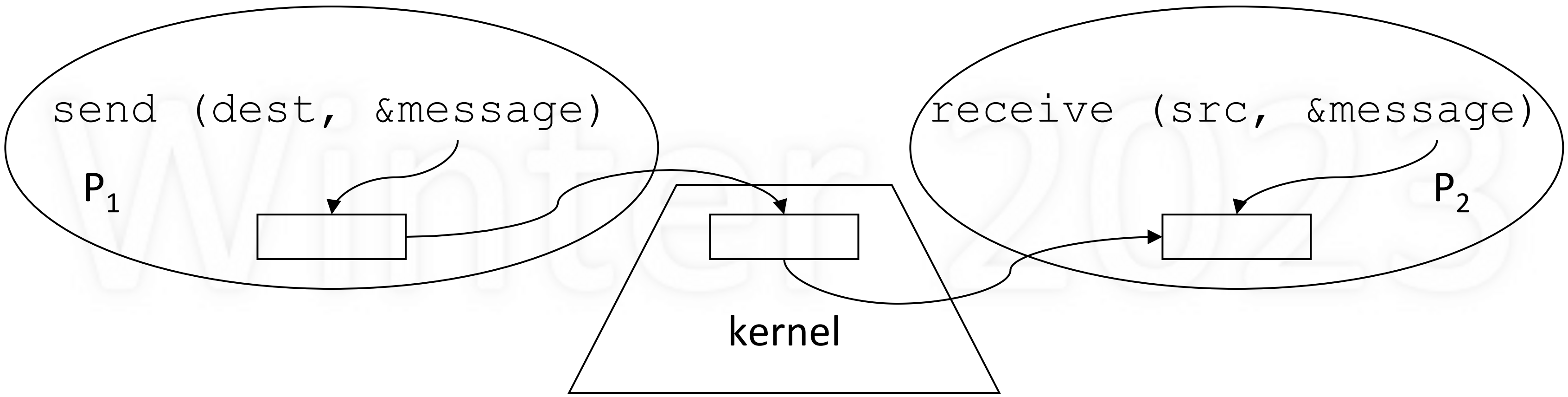
- open if no process *active* in monitor
- closes when process enters
- opens when process waits or exits
  (unless just signaled a waiting process)



Issues with Monitors

- Given P_1 waiting on condition c, P_2 signals c
  - P$_1$ and P_2 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



- Two methods
  - `send (destination, &message)`
  - `receive (source, &message)`
- Data transfer: in to and out of kernel message buffers
- Synchronization: receive blocks to wait for message

Producer/Consumer: Message-Passing

```
/* NO SHARED MEMORY */
```

<u>Producer</u>

```
int item;

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

<u>Consumer</u>

```
int item;

while (TRUE) {
    receive (Producer, &item);
    Consume (item);
}
```

With Flow Control

Producer

```
int item;

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

Consumer

```
int item;

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

while (TRUE) {
    receive (Producer, &item);
    Consume (item);
    send (Producer, &item);
}
```

An Optimization

Producer

```
int item, empty;




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

Consumer

```
int item, empty;

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

while (TRUE) {
    receive (Producer, &item);
    send (Producer, &empty);
    Consume (item);
}
```

Issues with Message Passing

- Who should messages be addressed to?
  - ports ("mailboxes") rather than processes
- How to make process receive from anyone?
  - `pid = receive (*, &message)`
- Kernel buffering: outstanding messages
  - messages sent that haven't been received yet
- Good paradigm for IPC over networks
- Safer than shared memory paradigms

Textbook

- OSP: Chapter 6
- OSC: Chapters 3, 6 (Process Synchronization)
  - Focus on semaphores, monitors, message-passing
  - Lecture-related: 3.4, 6.6, 6.7
  - Recommended: 3.5, 3.6