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

Prof. Joseph Pasquale
University of California, San Diego

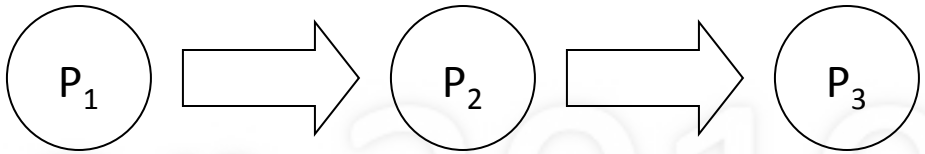
January 28, 2019

Cooperating Processes

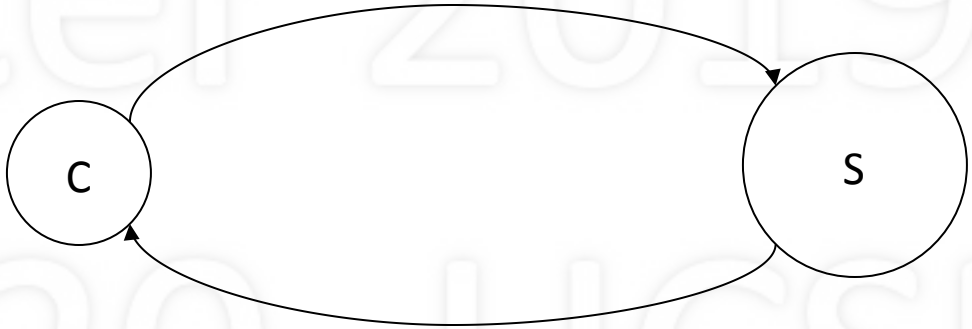
- Why structure a computation as a set of cooperating (communicating) processes?
- Performance: speed
  - Exploit inherent parallelism of computation
  - Allow some parts to proceed why others do I/O
- Modularity: reusable self-contained programs
  - Each may do a useful task on its own
  - May also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

Examples of Cooperating Processes

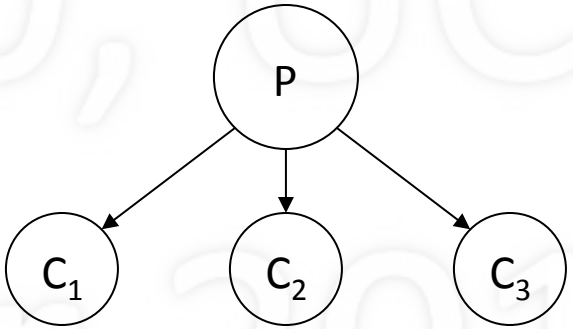
Pipeline



Client/Server



Parent/Child



Inter-Process Communication

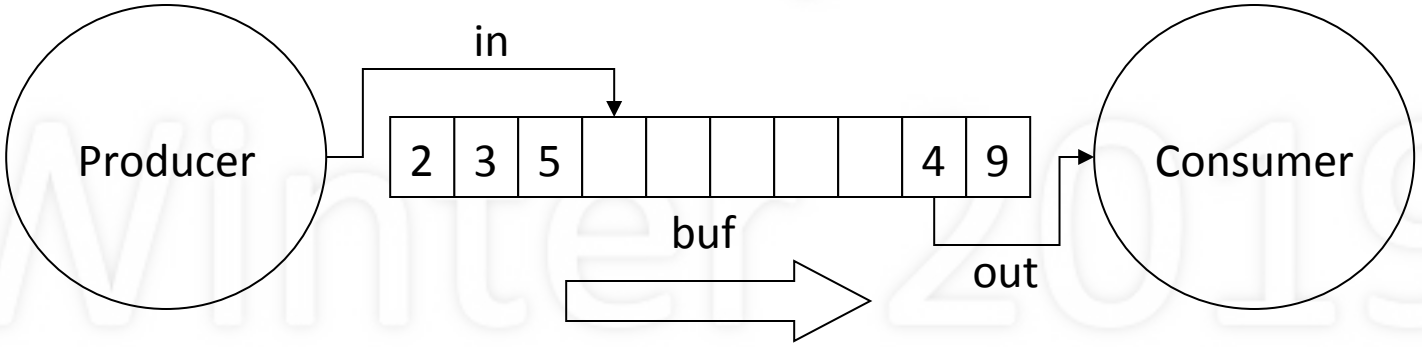
- To cooperate, need ability to communicate
- IPC: inter-process communication
  - Communication between processes
- IPC requires
  - *data transfer*
  - *synchronization*
- Need mechanisms for both

Three Abstractions for IPC

- Shared memory + semaphores
- Monitors
- Message passing

CSE 120, UCSD Winter 2019
CSE 120, UCSD Winter 2019

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                          Consumer
while (TRUE) {                    while (TRUE) {
    buf[in] = Produce ();             Consume (buf[out]);
    in = (in + 1)%N;                  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: easily leads to bugs
  - Example: what if wait statements are interchanged?

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);
    }
```

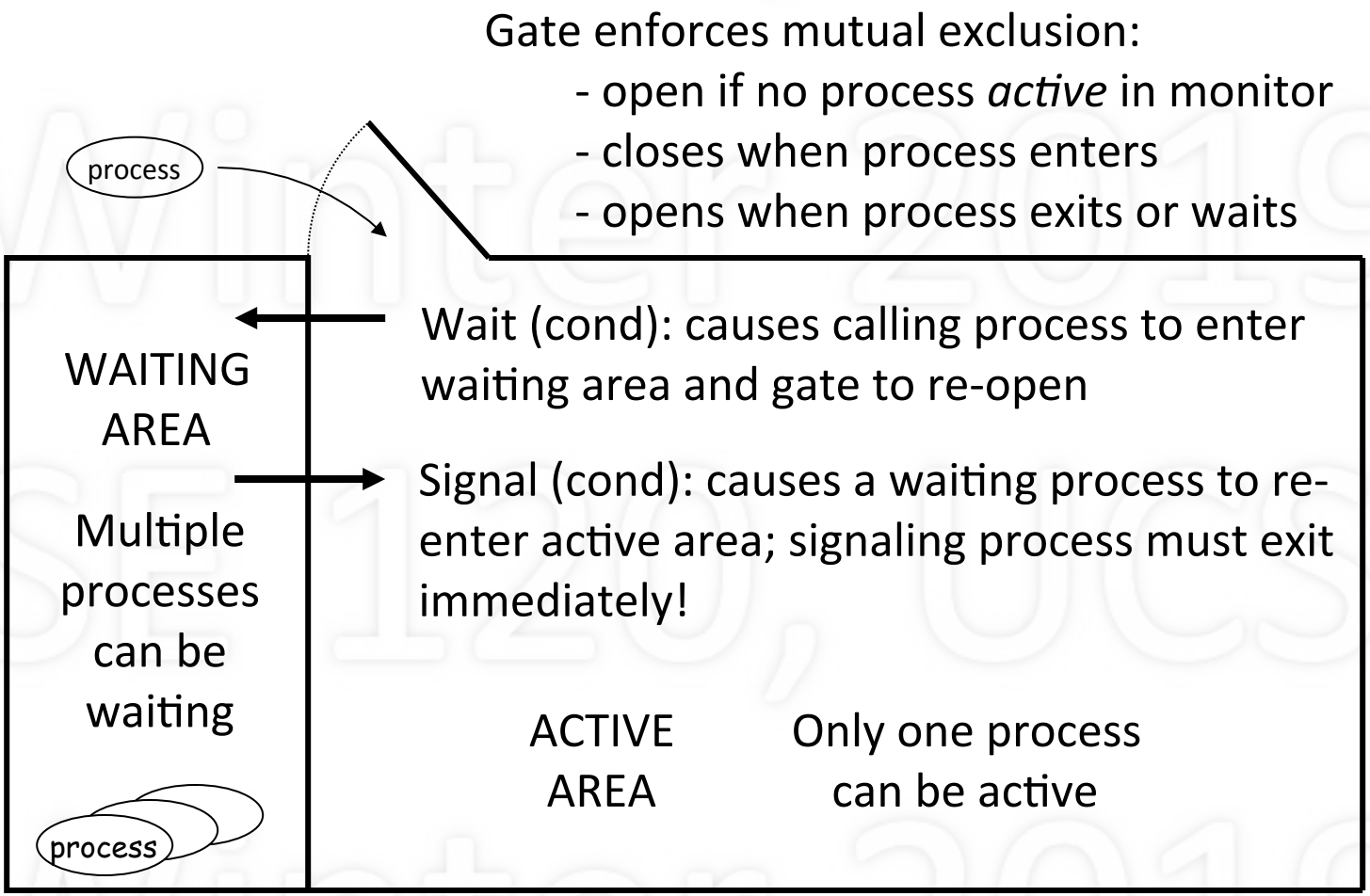
Producer

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

Consumer

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

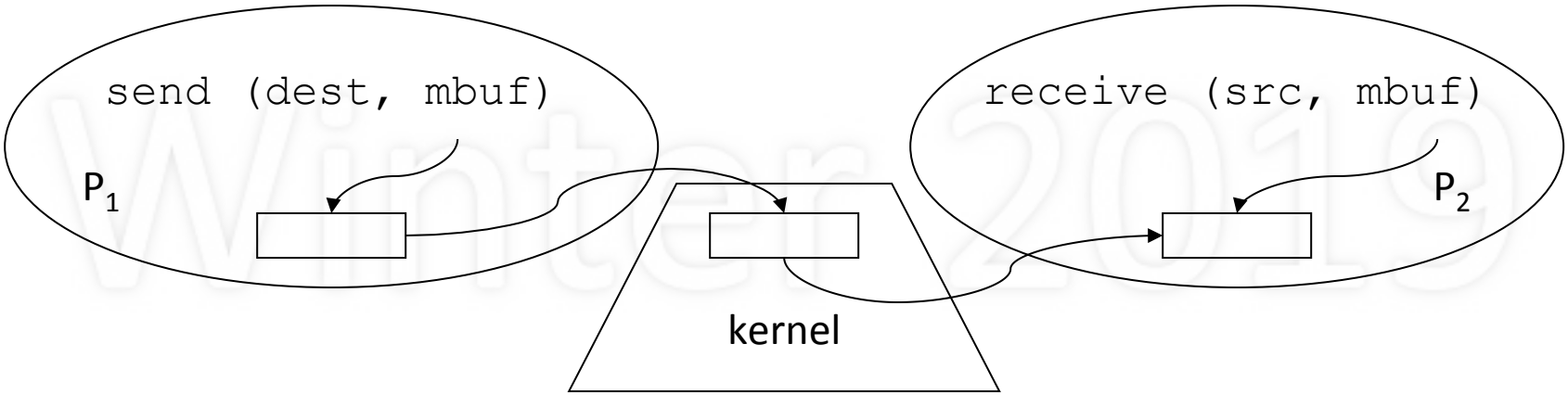
How Synchronization Works



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_buffer)`
  - `receive (source, message_buffer)`
- Data transfer: in to and out of kernel message buffers
- Synchronization: receive blocks to wait for message

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

```
int item, ready;

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

Consumer

```
int item, ready;

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

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

An Optimization

Producer

```
int item, ready;


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

Consumer

```
int item, ready;

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

while (TRUE) {
    receive (Producer, &item);
    send (Producer, &ready);
    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 (*, msg)`
- Kernel buffering: outstanding messages
  - messages sent that haven't been received yet
- Good paradigm for IPC over networks
- Safer than shared memory paradigms

Textbook

- Chapters 3, 5 (Process Synchronization)
  - Focus on semaphores, monitors, message-passing
  - Lecture-related: 3.4, 5.6, 5.8 (3.4, 6.5, 6.7 in 8th)
  - Recommended: 3.5, 3.6, 5.9 (3.5, 3.5, 6.8 in 8th)

Review & Research

- What is meant by processes that "cooperate"?
- Why structure a computation as a set of cooperating processes?
- What is meant by "inherent parallelism" of a computation?*
- How does parallelism promote performance?*
- Can you give an example of how cooperating processes promote modularity?**

R&R

- Given a single CPU, can a computation composed of cooperating processes run faster than the same computation structured as a single process (all single-threaded)?***
- Given cooperating processes, what are meant by the following structures or relationships: pipeline? client/server? parent/child? Can you give actual examples of each?*

R&R

- What is IPC?
- What are the two mechanisms required by IPC, and what is meant by each of them?
- Can you give an example of a mechanism that only achieves data transfer?* What about only synchronization?*

R&R

- What is the producer/consumer problem?
- In slide 6, how is data transfer implemented?*
- Why is synchronization needed?**
- In slide 7, what is the data transfer mechanism (or abstraction)?**
- Is there synchronization: why or why not?**
- Is synchronization actually needed, and if so, for what?**

R&R

- In slide 7, are there critical sections?***
- At the bottom of slide 7, what is meant by "Relevant if multiple producers or multiple consumers"?***
- In slide 9, can you explain the role of all the semaphores?**
- What is meant by "general synchronization vs. mutual exclusion"?**

R&R

- In slide 10, where is the race condition?**
- In slide 10, can you explain the role of all the semaphores?**
- In slide 11, what might happen if the wait statements in the Producer are interchanged? *** What about in the consumer?***

R&R

- What is a monitor?
- What are the elements that comprise monitors?
- How is data transfer achieved in monitors?**
- How is synchronization achieved in monitors? **
- How is mutual exclusion implemented in monitors?***

R&R

- What are condition variables in monitors, and how are they used?**
- What does the wait(cond) operation do?*
- What does the signal(cond) operation do?*
- What is meant by the property that “only one process can be active inside a monitor”?* How is this property implemented?***

R&R

- Can you explain how the code in slide 13 works?**
- In slide 13, how is data transfer achieved?**
- In slide 13, how mutual exclusion achieved?**
- In slide 13, how general synchronization achieved?**
- In slide 13, are there any system calls?***

R&R

- In slide 14, what is meant by the “gate”, and how does it work?**
- What is meant by the “Active Area”?* How about the “Waiting Area”?*
- How does a process enter the Active Area?* How about exit the Active Area?*
- What causes a process to enter the Waiting Area?* How about exit the “Waiting Area”?**

R&R

- What is the issue/problem with monitors that arises when a process P1 is waiting on condition c and another process signals c?**
- What is a solution to the problem?***
- What is meant by “Condition variables have no memory”?**
- If cond has no memory, then what is being supplied to wait(cond) and signal (cond)?***

R&R

- In slide 15, what is meant by “Signal is ‘lost’”? **
- How are monitors different from semaphores?***
- How are monitors different from semaphores + shared memory?***

R&R

- Can monitors be implemented with just semaphores?***
- Can semaphores be implemented with monitors?***
- What is meant by “power” in the statement: “monitors are more powerful than semaphores”?***

R&R

- What is message passing?
- What is the data transfer mechanism?
- What is the synchronization mechanism?*
- Can you explain how send (dest, mbuf) works? **
- Can you explain how receive (dest, mbuf) works?**
- What is the role of the kernel in the above?*

R&R

- Can you explain how the code in slide 17 works?**
- Are there system calls in slide 17?**
- Is it necessary that send (dest, mbuf) and receive (dest, mbuf) be atomic?***
- Can you explain how the code in slide 18 works?***
- What is the optimization in slide 19?***

R&R

- What is the issue with message passing regarding who (or what) should messages be addressed to?***
- What are ports (or mailboxes)?**
- Why is it convenient to allow processes to receive messages from anyone?***
- What is the issue with kernel buffering?***

R&R

- Why is message passing a good paradigm for IPC over networks?**
- In slide 20, meant by “shared memory paradigms”?***
- Why is message passing safer than shared memory paradigms?***

R&R

- Is message passing more powerful than monitors?***
- Is message passing more powerful than semaphores?***
- Can semaphores be implemented with monitors, and if so, how?***
- Can semaphores be implemented with message passing, and if so, how?***