3장. 프로세스 (2)

운영체제

목차

1. Process Concept
2. Process Scheduling
3. Operations on Processes
4. Interprocess Communication
5. **IPC in Shared-Memory Systems**
6. **IPC in Message-Passing Systems**
7. Examples of IPC Systems
8. Communication in Client-Server Systems

3.5 IPC IN SHARED-MEMORY SYSTEMS

3. Process

Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the **users processes** not the operating system.
- Major issues is to provide mechanism that will allow the user processes to **synchronize** their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
     . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Producer Process – Shared Memory

```
item next_produced;

while (true) {

        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */

        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
        while (in == out)
                ; /* do nothing */

        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

Bounded-Buffer – Shared-Memory Solution

- Solution is correct, but can only use BUFFER_SIZE-1 elements

What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer `counter` that keeps track of the number of full buffers.
- Initially, `counter` is set to 0.
- The integer `counter` is incremented by the producer after it produces a new buffer.
- The integer `counter` is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
          counter--;
        /* consume the item in next consumed */
}
```

Race Condition

- counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race Condition (cont.)

- Question
  - why was there no race condition in the first solution (where at most N – 1) buffers can be filled?
- More in Chapter 6.

3.6 IPC IN MESSAGE-PASSING SYSTEMS

3. Process

IPC – Message Passing

- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message)
  - receive(message)
- The message size is either fixed or variable

Message Passing (cont.)

- If processes P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive

Message Passing (cont.)

- Implementation issues:
  - **How** are links established?
  - Can a link be associated with **more than two** processes?
  - **How many links** can there be between every pair of communicating processes?
  - What is the **capacity** of a link?
  - Is the size of a message that the link can accommodate **fixed** or **variable**?
  - Is a link **unidirectional** or **bi-directional**?

Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus
  - Network

- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - send (P, message) – send a message to process P
  - receive(Q, message) – receive a message from process Q

- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from **mailboxes** (also referred to as **ports**)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox

- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication (cont.)

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox

- Primitives are defined as:
  - send(A, message) – send a message to mailbox A
  - receive(A, message) – receive a message from mailbox A

Indirect Communication (cont.)

- Mailbox sharing
  - P1, P2, and P3 share mailbox A
  - P1, sends; P2 and P3 receive
    - Who gets the message?

Indirect Communication (cont.)

- Mailbox sharing
  - P1, P2, and P3 share mailbox A
  - P1, sends; P2 and P3 receive
    - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

3.6.2 Synchronization

- Message passing may be either **blocking** or **non-blocking**
- Blocking is considered **synchronous**
  - Blocking send -- the sender is blocked until the message is received
  - Blocking receive -- the receiver is blocked until a message is available
- Non-blocking is considered **asynchronous**
  - Non-blocking send -- the sender sends the message and continue
  - Non-blocking receive -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Producer-Consumer: Message Passing

- Producer

```
message next_produced;
while (true) {
    /* produce an item in next_produced */

    send(next_produced);
}
```

- Consumer

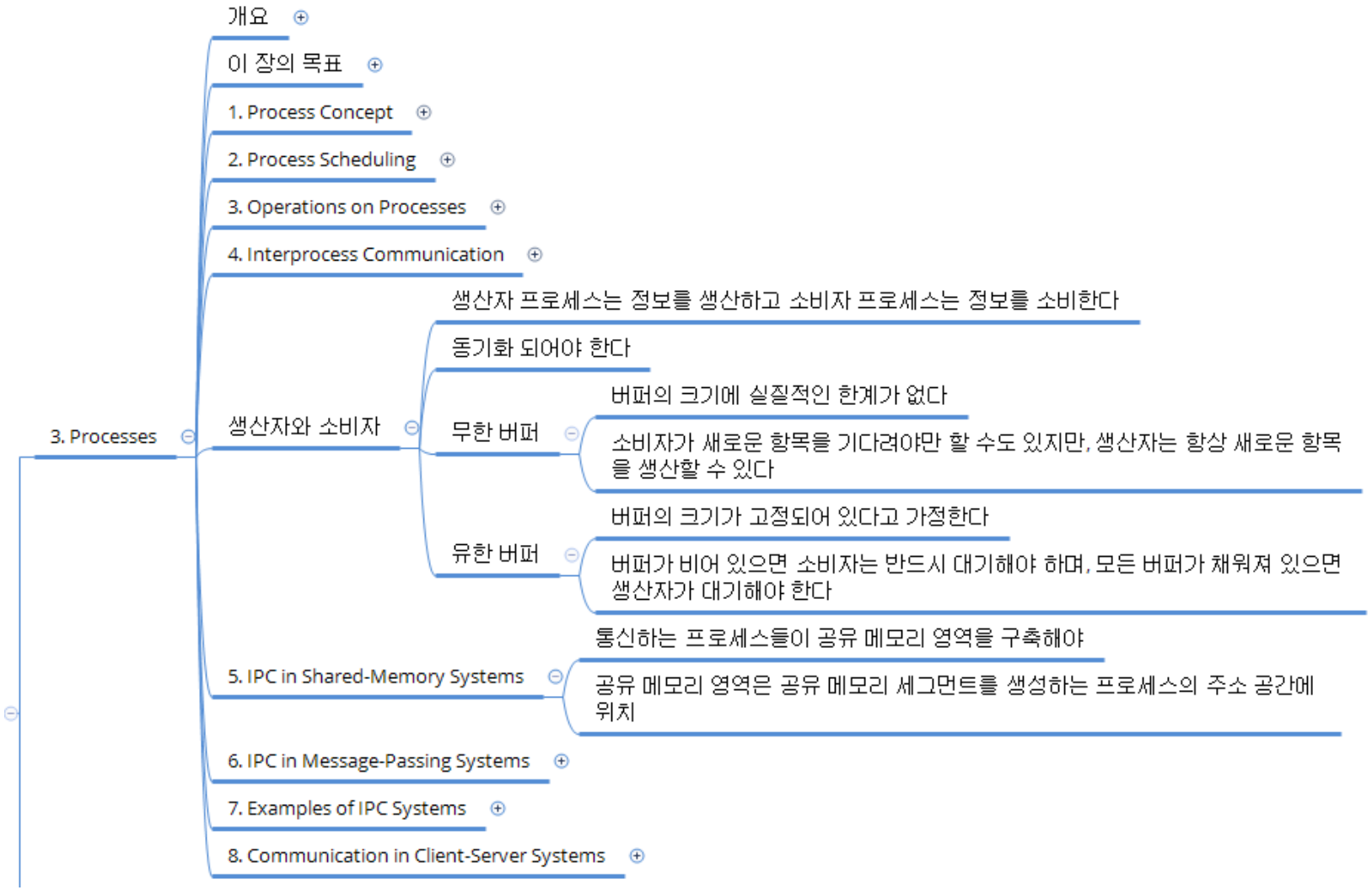
```
message next_consumed;
while (true) {
     receive(next_consumed)

    /* consume the item in next_consumed */
}
```

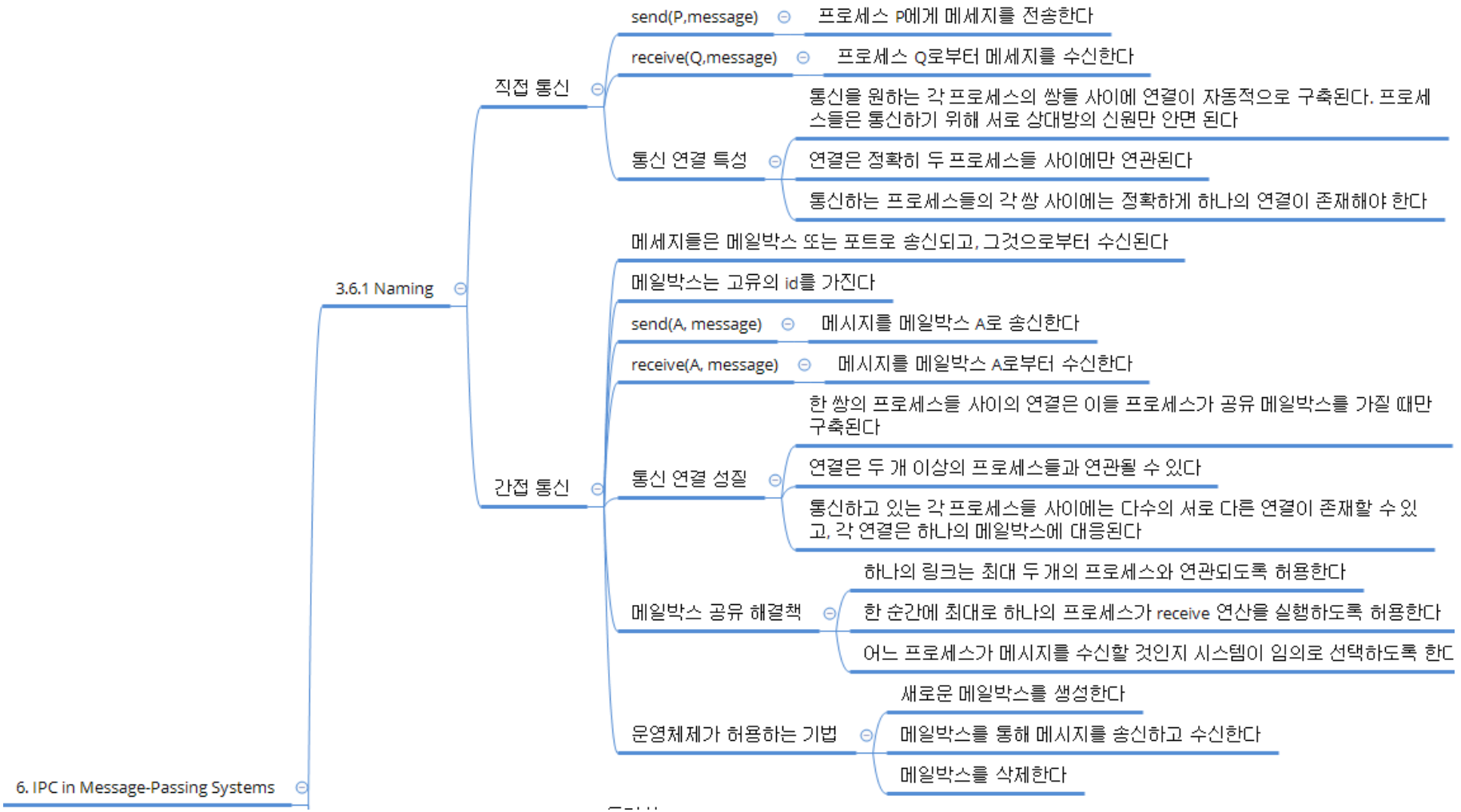
3.6.3 Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  - Zero capacity – no messages are queued on a link.
    Sender must wait for receiver (rendezvous)
  - Bounded capacity – finite length of n messages
    Sender must wait if link full
  - Unbounded capacity – infinite length
    Sender never waits

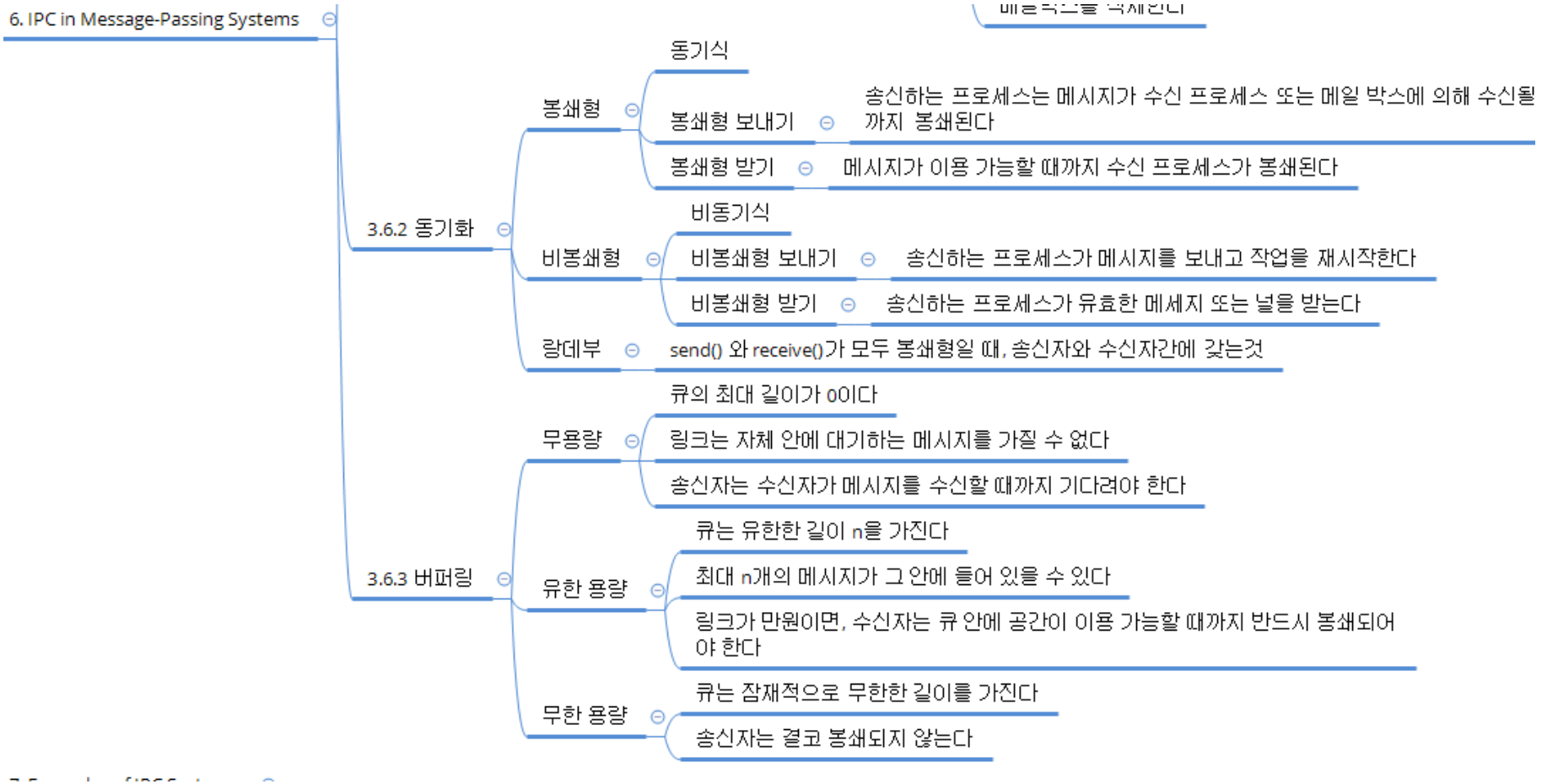
학습한 내용


3. Processes
개요
이 장의 목표
1. Process Concept
2. Process Scheduling
3. Operations on Processes
4. Interprocess Communication
생산자와 소비자
생산자 프로세스는 정보를 생산하고 소비자 프로세스는 정보를 소비한다
동기화 되어야 한다
무한 버퍼
버퍼의 크기에 실질적인 한계가 없다
소비자가 새로운 항목을 기다려야만 할 수도 있지만, 생산자는 항상 새로운 항목을 생산할 수 있다
유한 버퍼
버퍼의 크기가 고정되어 있다고 가정한다
버퍼가 비어 있으면 소비자는 반드시 대기해야 하며, 모든 버퍼가 채워져 있으면 생산자가 대기해야 한다
5. IPC in Shared-Memory Systems
통신하는 프로세스들이 공유 메모리 영역을 구축해야
공유 메모리 영역은 공유 메모리 세그먼트를 생성하는 프로세스의 주소 공간에 위치
6. IPC in Message-Passing Systems
7. Examples of IPC Systems
8. Communication in Client-Server Systems

학습한 내용


6. IPC in Message-Passing Systems
3.6.1 Naming
직접 통신
send(P,message)
프로세스 P에게 메세지를 전송한다
receive(Q,message)
프로세스 Q로부터 메세지를 수신한다
통신 연결 특성
통신을 원하는 각 프로세스의 쌍들 사이에 연결이 자동적으로 구축된다. 프로세스들은 통신하기 위해 서로 상대방의 신원만 안면 된다
연결은 정확히 두 프로세스들 사이에만 연관된다
통신하는 프로세스들의 각 쌍 사이에는 정확하게 하나의 연결이 존재해야 한다
간접 통신
메세지들은 메일박스 또는 포트로 송신되고, 그것으로부터 수신된다
메일박스는 고유의 id를 가진다
send(A, message)
메시지를 메일박스 A로 송신한다
receive(A, message)
메시지를 메일박스 A로부터 수신한다
통신 연결 성질
한 쌍의 프로세스들 사이의 연결은 이들 프로세스가 공유 메일박스를 가질 때만 구축된다
연결은 두 개 이상의 프로세스들과 연관될 수 있다
통신하고 있는 각 프로세스들 사이에는 다수의 서로 다른 연결이 존재할 수 있고, 각 연결은 하나의 메일박스에 대응된다
메일박스 공유 해결책
하나의 링크는 최대 두 개의 프로세스와 연관되도록 허용한다
한 순간에 최대로 하나의 프로세스가 receive 연산을 실행하도록 허용한다
어느 프로세스가 메시지를 수신할 것인지 시스템이 임의로 선택하도록 한다
운영체제가 허용하는 기법
새로운 메일박스를 생성한다
메일박스를 통해 메시지를 송신하고 수신한다
메일박스를 삭제한다

학습한 내용


6. IPC in Message-Passing Systems
3.6.2 동기화
봉쇄형
동기식
봉쇄형 보내기
송신하는 프로세스는 메시지가 수신 프로세스 또는 메일 박스에 의해 수신될 까지 봉쇄된다
봉쇄형 받기
메시지가 이용 가능할 때까지 수신 프로세스가 봉쇄된다
비봉쇄형
비동기식
비봉쇄형 보내기
송신하는 프로세스가 메시지를 보내고 작업을 재시작한다
비봉쇄형 받기
송신하는 프로세스가 유효한 메세지 또는 널을 받는다
랑데부
send() 와 receive()가 모두 봉쇄형일 때, 송신자와 수신자간에 갖는것
3.6.3 버퍼링
무용량
큐의 최대 길이가 0이다
링크는 자체 안에 대기하는 메시지를 가질 수 없다
송신자는 수신자가 메시지를 수신할 때까지 기다려야 한다
유한 용량
큐는 유한한 길이 n을 가진다
최대 n개의 메시지가 그 안에 들어 있을 수 있다
링크가 만원이면, 수신자는 큐 안에 공간이 이용 가능할 때까지 반드시 봉쇄되어야 한다
무한 용량
큐는 잠재적으로 무한한 길이를 가진다
송신자는 결코 봉쇄되지 않는다