3장: 프로세스

3장: 프로세스

프로세스 개념프로세스 스케줄링프로세스에 대한 작업프로세스 간 통신 IPC 시스템의 예클라이언트-서버 시스템에서의 통신

목표

모든 계산의 기초를 형성하는 실행 중인 프로그램인 프로세스의 개념을 소개합니다.스케줄링, 생성 및 종료, 커뮤니케이션을 포함한 프로세스의 다양한 기능을 설명합니다.공유 메모리 및 메시지 전달을 사용하여 프로세스 간 커뮤니케이션을 탐구합니다.클라이언트-서버 시스템에서의 커뮤니케이션을 설명합니다.

프로세스 개념

운영 체제는 다양한 프로그램을 실행합니다.

배치 시스템 – 작업시간 공유 시스템 – 사용자 프로그램 또는 작업교과서는 작업과 프로세스라는 용어를 거의 같은 의미로 사용합니다.프로세스 – 실행 중인 프로그램; 프로세스 실행은 순차적으로 진행되어야 합니다.여러 부분

텍스트 섹션이라고도 하는 프로그램 코드프로그램 카운터를 포함한 현재 활동, 프로세서 레지스터임시 데이터를 포함하는 스택

□ 함수 매개 변수, 반환 주소, 로컬 변수전역 변수를 포함하는 데이터 섹션런타임 동안 동적으로 할당된 메모리를 포함하는 힙

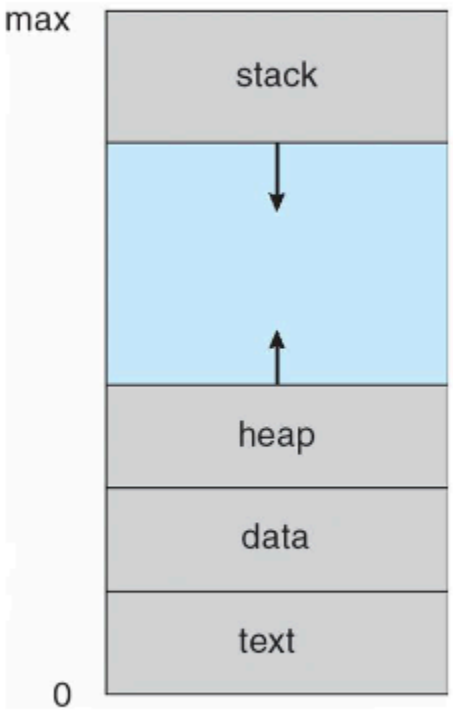
프로세스 개념 (계속)

프로그램은 디스크(실행 파일)에 저장된 수동 개체이고 프로세스는 활성 상태입니다.

실행 파일이 메모리에 로드되면 프로그램이 프로세스가 됩니다.GUI 마우스 클릭, 이름 입력의 명령줄 입력 등을 통해 시작된 프로그램 실행하나의 프로그램은 여러 프로세스가 될 수 있습니다.

여러 사용자가 동일한 프로그램을 실행한다고 가정합니다

메모리의 프로세스

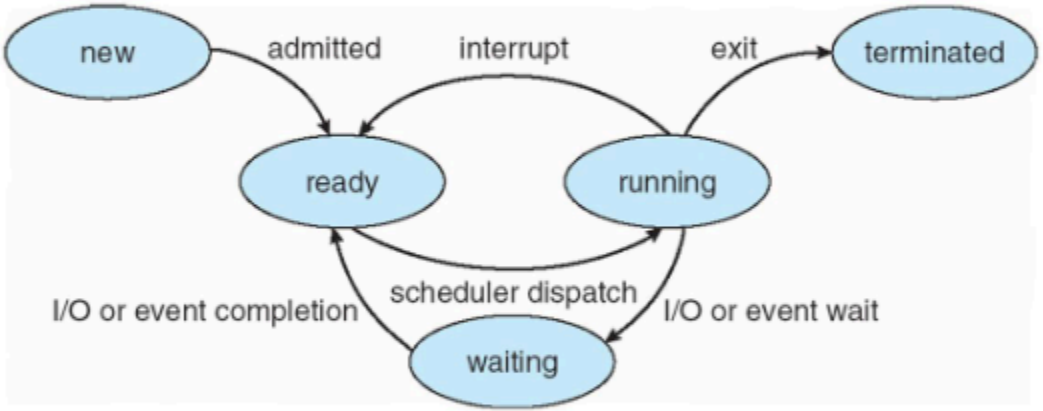


프로세스 상태

프로세스가 실행되면 상태가 변경됩니다

신규: 프로세스가 생성되고 있습니다running: 지침이 실행되고 있습니다waiting: 프로세스가 일부 이벤트가 발생하기를 기다리고 있습니다ready: 프로세스가 프로세서에 할당되기를 기다리고 있습니다 terminated: 프로세스가 실행을 완료했습니다.

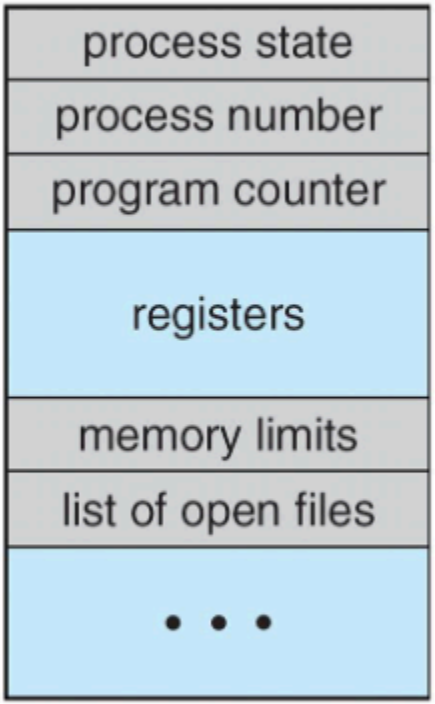
프로세스 상태 다이어그램



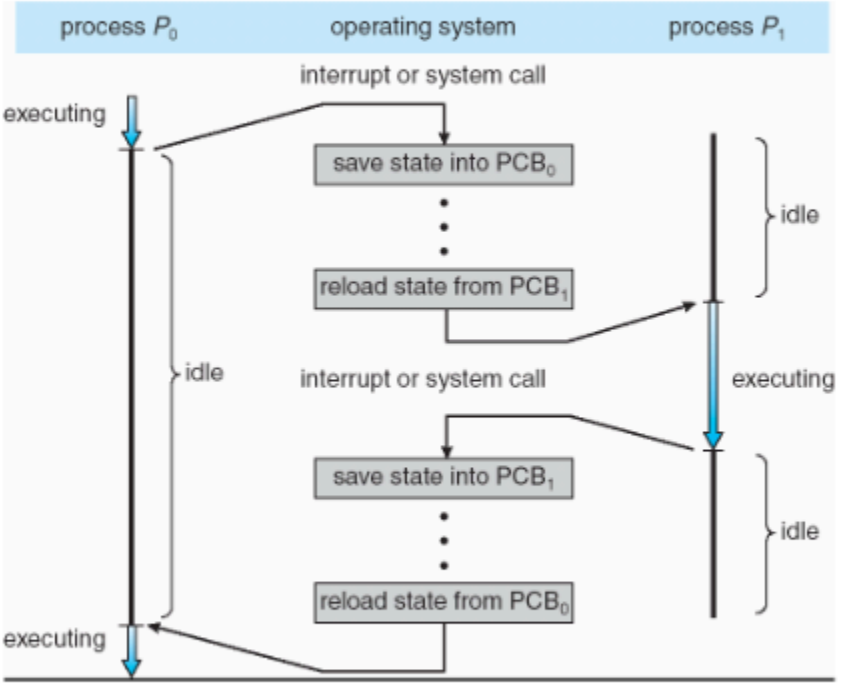
공정 제어 블록(PCB)

각 프로세스와 관련된 정보(작업 제어 블록이라고도 함)

프로세스 상태 – 실행 중, 대기 중 등프로그램 카운터 – 다음에 실행할 명령어의 위치 CPU 레지스터 – 모든 프로세스 중심 레지스터의 내용CPU 스케줄링 정보- 우선 순위, 스케줄링 큐 포인터메모리 관리 정보 – 프로세스에 할당된 메모리회계 정보 – 사용된 CPU, 시작 이후 경과된 클럭 시간, 시간 제한I/O 상태 정보 – I/O 장치 할당됨process, 열린 파일 목록



프로세스에서 프로세스로 CPU 전환



스레드

지금까지 프로세스에는 단일 실행 스레드가 있습니다.프로세스 당 여러 프로그램 카운터를 갖는 것을 고려하십시오.

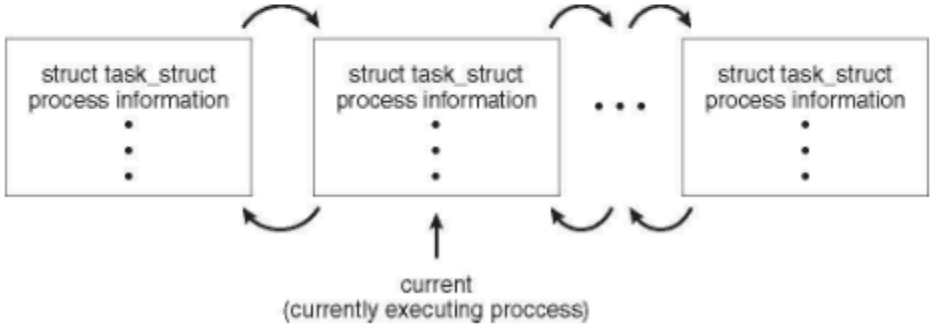
여러 위치에서 한 번에 실행할 수 있습니다.

- 제어 스레드의 여러 스레드 -> 스레드그런 다음 스레드 세부 정보, 여러 프로그램 카운터를 위한 저장소가 있어야 합니다.PCBSee 다음 장 참조

Linux의 프로세스 표현

C 구조체 task_struct로 표현됩니다.

```
PID t_pid; /* 프로세스 식별자 */long 상태; /* 프로세스의 상태
*/unsigned int time_slice /* 스케줄링 정보 */struct task_struct
*parent; /* 이 프로세스의 부모 */struct list_head 자식; /* 이 프로세
스의 자식 */struct files_struct *files; /* 열린 파일의 리스트
*/struct mm_struct *mm; /* 이 프로세스의 주소 공간 */
```

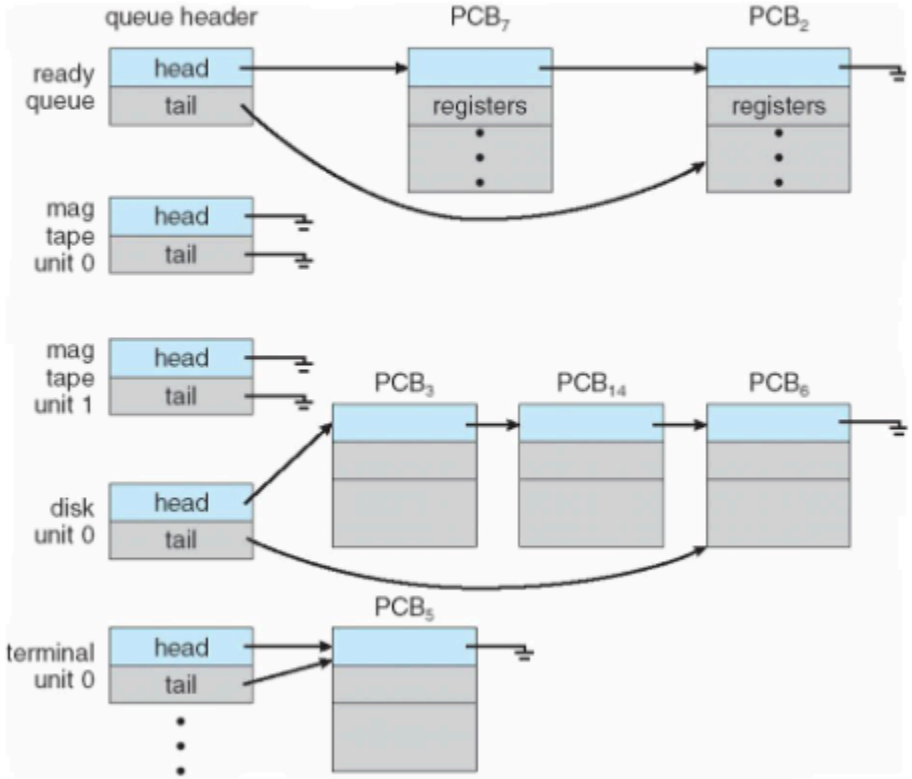


프로세스 스케줄링

CPU 사용 극대화, 시간 공유를 위해 프로세스를 CPU로 빠르게 전환프로세스 스케줄러는 CPU에서 다음 실행을 위해 사용 가능한 프로세스 중에서 선택프로세스의 스케줄링 대기열을 유지합니다.

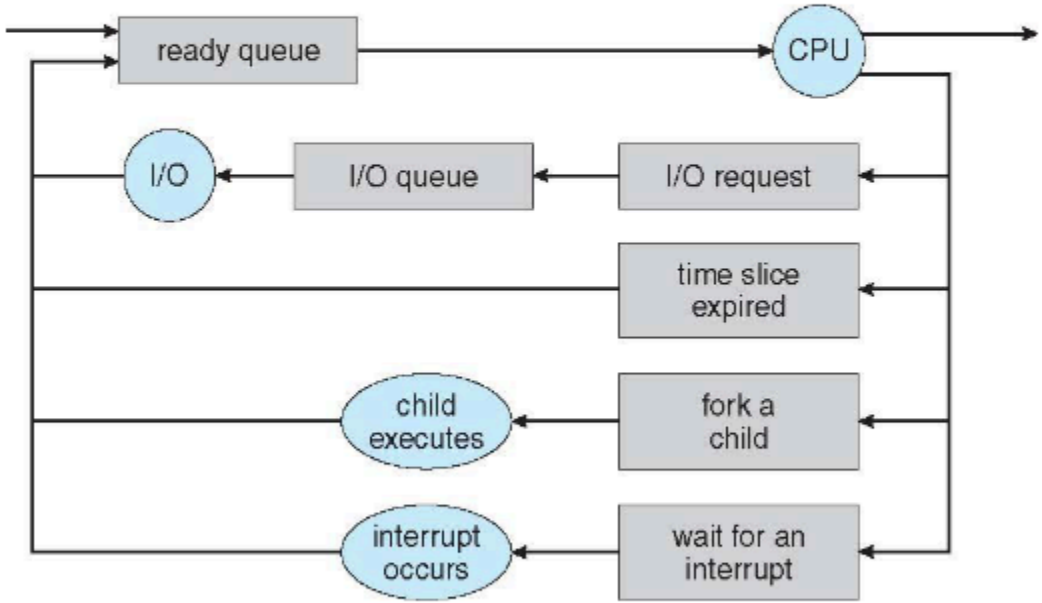
작업 큐 – systemReady 큐의 모든 프로세스 집합 – 주 메모리에 있는 모든 프로세스 집합, 준비 완료 및 실행 대기 장치 큐 – I/O 장치를 기다리는 프로세스 집합다양한 큐 간에 프로세스 마이그레이션

Ready Queue 및 다양한 I/O Device Queue



프로세스 스케줄링의 표현

대기열 다이어그램은 대기열, 리소스, 흐름을 나타냅니다



스케줄러

단기 스케줄러(또는 CPU 스케줄러) – 다음에 실행해야 하는 프로세스를 선택하고 CPU를 할당합니다.

때때로 시스템의 유일한 스케줄러단기 스케줄러가 자주 호출됩니다 (밀리초) ㅁ (빨라야 함)장기 스케줄러 (또는 작업 스케줄러) – 준비 대기열로 가져와야 하는 프로세스를 선택합니다.

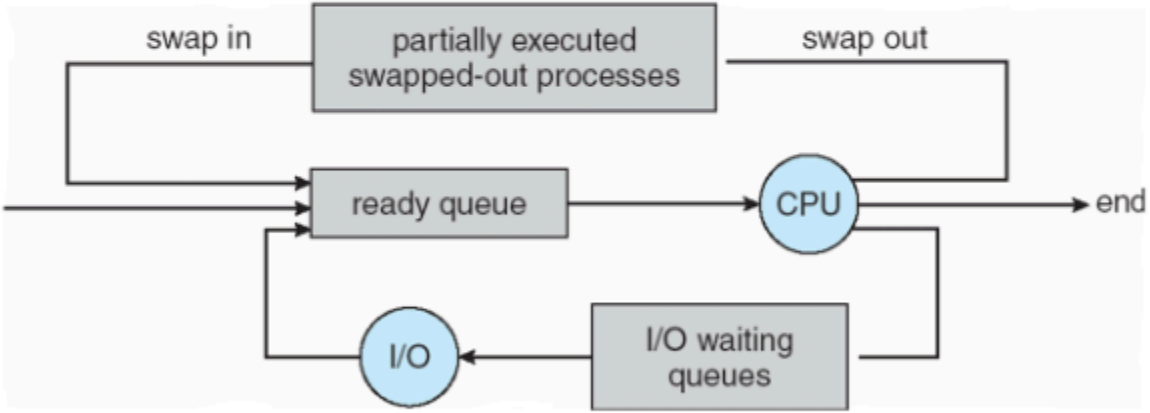
장기 스케줄러는 드물게 호출됩니다 (초, 분) ㅁ (느려질 수 있음)장기 스케줄러는 멀티 프로그래밍의 정도를 제어합니다.프로세스는 다음 중 하나로 설명 할 수 있습니다.

I/O 바운드 프로세스 – 계산보다 I/O에 더 많은 시간을 소비함, 많은 짧은 CPU 버스트CPU 바운드 프로세스 – 계산에 더 많은 시간을 소비합니다. 매우 긴 CPU 버스트 적음장기 스케줄러는 좋은 프로세스 혼합을 위해 노력합니다.

중기 스케줄링 추가

다중 프로그래밍의 정도를 줄여야 하는 경우 중기 스케줄러를 추가할 수 있습니다.

메모리에서 프로세스 제거, 디스크에 저장, 디스크에서 다시 가져와 실행 계속: 스와핑



모바일 시스템의 멀티태스킹

일부 모바일 시스템(예: iOS의 초기 버전)은 하나의 프로세스만 실행할 수 있고 다른 시스템은 일시 중단화면 공간으로 인해 iOS가 제공하는 사용자 인터페이스 제한으로 인해

단일 포그라운드 프로세스 - 사용자 인터페이스를 통해 제어됨여러 백그라운드 프로세스 – 메모리에 있고 실행 중이지만 디스플레이에 표시되지 않으며 제한이 있음제한에는 단일, 짧은 작업, 이벤트 알림 수신, 특정 오디오 재생과 같은 장기 실행 작업이 포함됩니다.Android는 더 적은 제한으로 포그라운드 및 백그라운드를 실행합니다.

백그라운드 프로세스는 서비스를 사용하여 작업을 수행함백그라운드 프로세스가 일시 중단되어도 서비스를 계속 실행할 수 있음서비스에 사용자 인터페이스가 없고 메모리를 적게 사용합니다.

컨텍스트 전환

CPU가 다른 프로세스로 전환될 때 시스템은 이전 프로세스의 상태를 저장하고 컨텍스트 switch를 통해 새 프로세스에 대해 저장된 상태를 로드해야 합니다.PCBContext-switch 시간에 표시되는 프로세스의 컨텍스트는 오버헤드입니다. 전환하는 동안 시스템은 유용한 작업을 수행하지 않습니다.

OS와 PCB가 복잡할수록 하드웨어 지원에 의존하는 컨텍스트 switchTime이 길어집니다.

- 일부 하드웨어는 CPU 당 여러 레지스터 세트를 제공합니다 ➔ 한 번에 로드되는 다중 컨텍스트

프로세스에 대한 작업

시스템은 다음을 위한 메커니즘을 제공해야 합니다.

프로세스 생성, 프로세스 종료 등 다음에 자세히 설명되어 있습니다

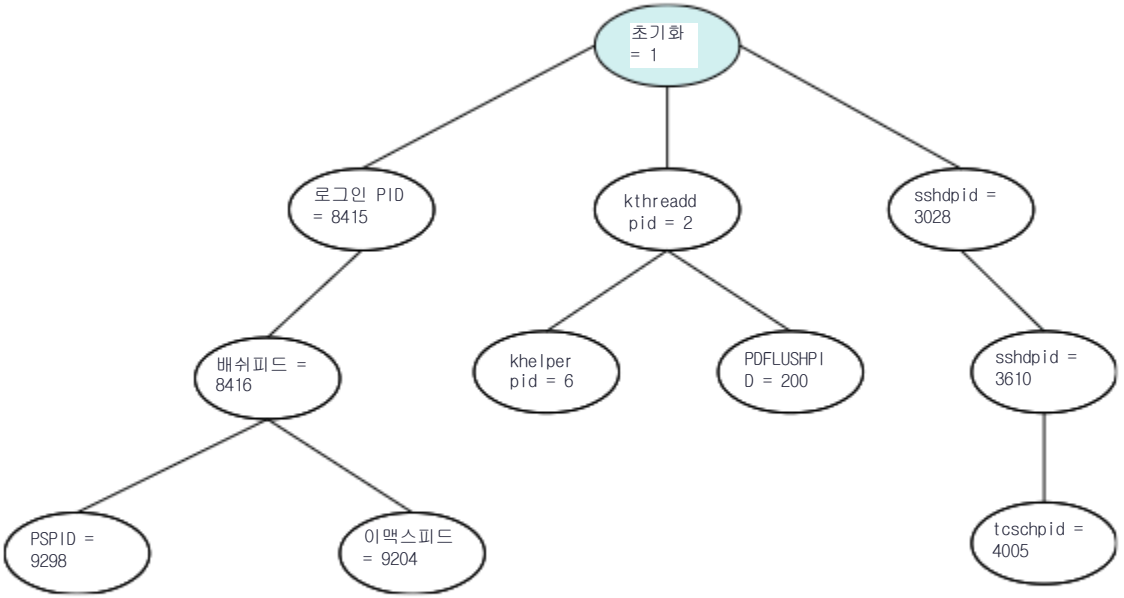
프로세스 생성

Parent process는 자식 프로세스를 생성하고, 이 프로세스는 다시 다른 프로세스를 생성하고,프로세스 트리를 형성합니다.일반적으로 프로세스 식별자(pid)를 통해 식별되고 관리되는 프로세스리소스 공유 옵션

부모와 자식은 모든 리소스를 공유합니다
자식은 부모의 리소스 하위 집합을 공유합
니다부모와 자식은 리소스를 공유하지 않
습니다.실행 옵션

부모와 자식이 동시에 실행부모는
자식이 종료될 때까지 기다립니다

Linux의 프로세스 트리

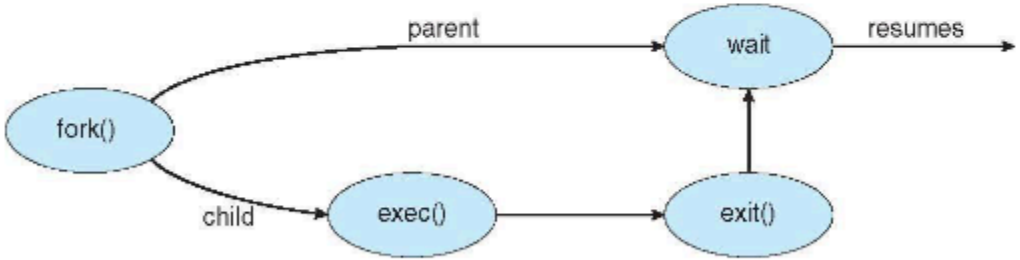


프로세스 생성(계속)

주소 공간

parentChild의 자식 중복에는 프로그램이 로드되어 있습니다.UNIX 예제 fork() 시스템 호출은 새 프로세스를 생성합니다.

exec() 시스템 호출은 fork() 다음에 사용되어 프로세스의 메모리 공간을 새 프로그램으로 교체합니다.



C 프로그램 포킹 개별 프로세스

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

Windows API를 통해 별도의 프로세스 만들기

```
int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   /* allocate memory */
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   /* create child process */
   if (!CreateProcess(NULL, /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
   {
     fprintf(stderr, "Create Process Failed");
     return -1;
   }
   /* parent will wait for the child to complete */
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   /* close handles */
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

프로세스 종료

Process는 마지막 문을 실행한 다음 exit()system 호출을 사용하여 운영 체제에 해당 문을 삭제하도록 요청합니다.

자식에서 부모로 상태 데이터 반환 (wait() 를 통해) Process의 리소스는 운영 체제에 의해 할당 해제됩니다 Parent는 abort () 시스템 호출을 사용하여 자식 프로세스의 실행을 종료 할 수 있습니다. 그렇게 하는 몇 가지 이유는 다음과 같습니다.

자식이 할당된 리소스를 초과했습니다자식에 할당된 작업이 더 이상 필요하지 않습니다.부모가 종료되고 있으며 운영 체제는 부모가 종료되는 경우 자식이 계속할 수 없도록 합니다.

프로세스 종료

일부 운영 체제에서는 부모가 종료된 경우 자식이 존재할 수 없습니다. 프로세스가 종료되면 모든 자식도 종료되어야 합니다.

계단식 종료. 모든 자녀, 손자 등이 중절됩니다. 종료는 운영 체제에 의해 시작됩니다. 부모 프로세스는 wait() 시스템 호출을 사용하여 자식 프로세스의 종료를 기다릴 수 있습니다. 호출은 상태 정보와 종료된 프로세스의 pid를 반환합니다

pid = 대기(&상태); 부모가 대기하지 않는 경우(wait()를 호출하지 않음) 프로세스가 좀비인 경우부모가 wait를 호출하지 않고 종료된 경우 프로세스는 고아입니다.

멀티프로세스 아키텍처 – Chrome 브라우저

많은 웹 브라우저가 단일 프로세스로 실행되었습니다 (일부는 여전히 실행됨)

하나의 웹 사이트가 문제를 일으키면 전체 브라우저가 중단되거나 충돌 할 수 있습니다.Google Chrome 브라우저는 3 가지 유형의 프로세스가있는 다중 프로세스입니다.

브라우저 프로세스는 사용자 인터페이스, 디스크 및 네트워크를 관리합니다 I / ORenderer 프로세스는 웹 페이지를 렌더링하고 HTML, Javascript를 처리합니다. 열린 각 웹 사이트에 대해 생성된 새 렌더러

- 디스크 및 네트워크 I/O를 제한하는 샌드박스에서 실행되어 영향을 최소화합니다.

보안 악용각 플러그인 유형에 대한
플러그인 프로세스

프로세스 간 통신

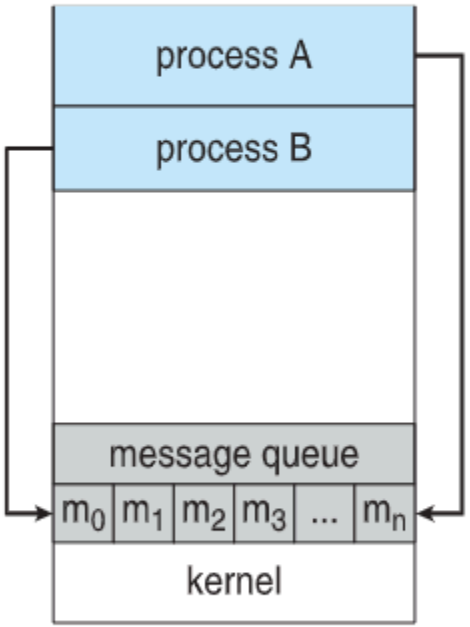
시스템 내의 프로세스는 독립적이거나 협력적일 수 있습니다.협력 프로세스는 데이터 공유를 포함하여 다른 프로세스에 영향을 미치거나 영향을 받을 수 있습니다.협력 프로세스의 이유:

정보 공유계산 속도 향상모듈성편의성협력 프로세스에는 IPC(Interprocess Communication)가 필요합니다IPC의 두 가지 모델

공유 메모리메
시지 전달

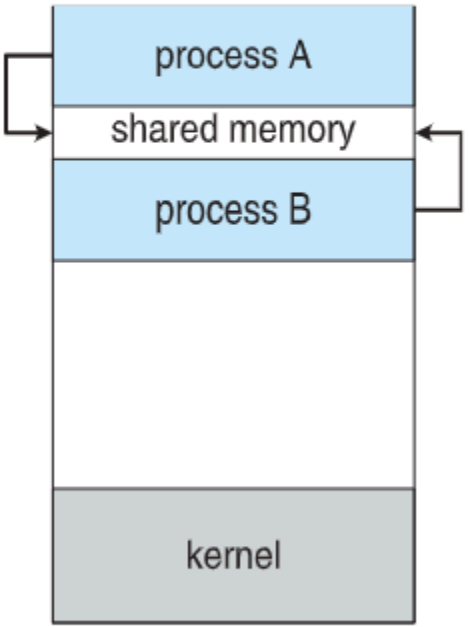
통신 모델

(a) 메시지 전달.



(a)

(b) 공유 메모리.



(b)

협력 프로세스

독립적 인 프로세스는 다른 프로세스의 실행에 영향을 미치거나 영향을 받을 수 없습니다.협력 프로세스는 다른 프로세스의 실행에 영향을 미치거나 영향을받을 수 있습니다.프로세스 협력의 장점

정보 공유계산 속도
향상
ModularityConveni
ence

생산자-소비자 문제

협력 프로세스의 패러다임, 생산자 프로세스는 소비자 프로세스에 의해 소비되는 정보를 생산합니다

unbounded-buffer 는 고정 된 버퍼 크기가 있다고 가정하여 bufferbounded-buffer 의 크기에 실질적인 제한을 두지 않습니다

bounded-buffer – 공유 메모리 솔루션

공유 데이터

```
#define BUFFER_SIZE
10typedef 구조체 {
. . .}
항목;

항목 버퍼[BUFFER_SIZE]; int in
= 0; int 출력 = 0;
```

솔루션은 정확하지만 BUFFER_SIZE-1 요소만 사용할 수 있습니다.

bounded-buffer – 생산자

```
항목 next_produced;
while (참) {
        /* 다음에 생성된 항목에서 항목 생성 */while
        (((in + 1) % BUFFER_SIZE) == out)
                ; /* 아무것도하지 않습니다 */
        버퍼[in] = next_produced; in =
        (in + 1) % BUFFER_SIZE;
}
```

바운디드 버퍼 – 소비자

```
항목 next_consumed;
while (참) {
        while (입력 == 출력)
                ; /* 아무것도하지 않습니다 */
        next_consumed = 버퍼[출력]; 출력 =
        (출력 + 1) % BUFFER_SIZE;

        /* 다음에 소비한 아이템 소비 */
}
```

프로세스 간 통신 – 공유 메모리

통신하고자 하는 프로세스들 사이에서 공유되는 메모리 영역통신은 운영 체제가 아닌 사용자 프로세스의 제어 하에 있습니다. 주요 문제는 사용자 프로세스가 공유 메모리에 액세스할 때 작업을 동기화할 수 있도록 하는 메커니즘을 제공하는 것입니다. 동기화는 5장에서 자세히 설명합니다.

프로세스 간 통신 – 메시지 전달

프로세스가 통신하고 작업을 동기화하는 메커니즘메시지 시스템 – 프로세스는 공유 변수에 의존하지 않고 서로 통신IPC 기능은 두 가지 작업을 제공합니다.

send(message)receive(message)메시지 크기는 고정되어 있거나 가변적입니다.

메시지 전달(계속)

프로세스 P와 Q가 통신하려면 다음을 수행해야 합니다.

그들 사이의 통신 링크 설정send/receive를 통
해 메시지 교환구현 문제:

링크는 어떻게 설정됩니까? 링크를 세 개 이상의 프로세스와 연결할 수 있습니까? 모든 통신 프로세스 쌍 사이에 얼마나 많은 링크가 있을 수 있습니까? 링크의 용량은 얼마입니까? 링크가 수용할 수 있는 메시지의 크기는 고정되어 있습니까, 아니면 가변적입니까? 링크는 단방향입니까, 양방향입니까?

메시지 전달(계속)

통신 링크 구현

육체의:

□ 공유 메모리□
하드웨어 버스□

NetworkLogical:

□ 직접 또는 간접 □ 동기식 또는 비동기식 □ 자동 또는 명시적 버퍼링

직접 통신

프로세스는 서로의 이름을 명시적으로 지정해야 합니다.

send (P, message) – 프로세스 Preceive(Q, message) – 통신 링크의 프로세스 QProperties에서 메시지를 수신합니다.

링크는 자동으로 설정됨링크는 정확히 한 쌍의 통신 프로세스와 연결됩니다각 쌍 사이에는 정확히 하나의 링크가 존재합니다.링크는 단방향일 수 있지만 일반적으로 양방향입니다.

간접 의사소통

메시지는 사서함(포트라고도 함)에서 전달되고 수신됩니다

각 사서함에는 고유한 idProcesses가 있습니다.통신 링크의 mailboxProperties를 공유하는 경우에만 통신할 수 있습니다.

프로세스가 공통 사서함을 공유하는 경우에만 링크가 설정됨링크는 여러 프로세스와 연결될 수 있음각 프로세스 쌍은 여러 통신 링크를 공유할 수 있음링크는 단방향 또는 양방향일 수 있음

간접 의사소통

작업

새 사서함 만들기(포트)사서함을 통해 메시지 보내기 및 받기사서함 파괴기본 형식은 다음과 같이 정의됩니다.send(A, message) – 사서함으로 메시지 보내기 Areceive(A, message) – 사서함 A에서 메시지 받기

간접 의사소통

사서함 공유

P1, P2 및 P3은 사서함 AP1을 공유합니다. P2 및 P3 수신누가 메시지를 받습니까? 솔루션

링크를 최대 두 개의 프로세스와 연결할 수 있도록 허용한 번에 하나의 프로세스만 수신 작업을 실행할 수 있도록 허용시스템이 수신자를 임의로 선택할 수 있도록 허용합니다. 발신자는 수신자가 누구인지 알림을 받습니다.

동기화

메시지 전달은 차단 또는 비 차단일 수 있습니다.차
단은 동기로 간주됩니다.

송신 차단 -- 메시지를 받을 때까지 발신자가 차단됩니다.수신 차단 -- 메시지를 사용할 수 있을 때까지 수신자가 차단됩니다.비차단은 비동기로 간주됩니다.

비차단 보내기 -- 발신자가 메시지를 보내고 계속비차단
수신 -- 수신자가 다음을 수신합니다.

유효한 메시지 또는Null 메
시지다양한 조합 가능

보내기와 받기가 모두 차단되면 랑데부가 있습니다

동기화(연속)

생산자-소비자는 사소해진다.

```
메시지 next_produced;
    while (참) {
    /* 다음에 생성된 항목에서 항목 생성
    */send(next_produced);}


메시지 next_consumed;
while (참) {
    수신(next_consumed);

    /* 다음에 소비한 아이템 소비 */
}
```

버퍼링

링크에 첨부된 메시지의 대기열.다
음 세 가지 방법 중 하나로 구현

1. 용량 0 – 링크에서 대기열에 메시지가 없습니다. 발신자는 수신자를 기다려야 함(랑데부)
2. 제한된 용량 – n개의 메시지의 한정된 길이발신자는 링크가 가득 차면 기다려야 합니다.
3. 무한한 용량 – 무한 길이Sender는 절대 기다리지 않습니다

IPC 시스템의 예 - POSIX

POSIX 공유 메모리

프로세스는 먼저 공유 메모리를 생성합니다 segmentshm_fd = shm_open(name, O CREAT | O RDWR, 0666); 또한 기존 세그먼트를 열어 공유하는 데 사용됩니다.개체의 크기를 설정합니다.

ftruncate(shm fd, 4096); 이제 프로세스는 공유 메모리에 쓸 수 있습니다.sprintf(shared memory, "Writing to shared memory");

IPC POSIX 프로듀서

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX 소비자

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

IPC 시스템의 예 - 마하

Mach 커뮤니케이션은 메시지 기반입니다.

시스템 호출도 메시지입니다.각 작업은 생성 시 두 개의 메일박스(Kernel 및 Notify)를 받습니다.메시지 전송에 필요한 시스템 호출은 세 개뿐입니다.

`msg_send(), msg_receive(), msg_rpc()`

통신을 위해 필요한 메일박스, 다음을 통해 생성

`port_allocate()`

보내기 및 받기는 유연하며, 예를 들어 사서함이 가득 찬 경우 네 가지 옵션이 있습니다.

□ 무기한 대기□ 최대 n밀리초 대기□ 즉시 반환□ 메시지를 임시로 캐시

IPC 시스템의 예 – Windows

고급 LPC(Local Procedure Call) 기능을 통한 메시지 전달 중심

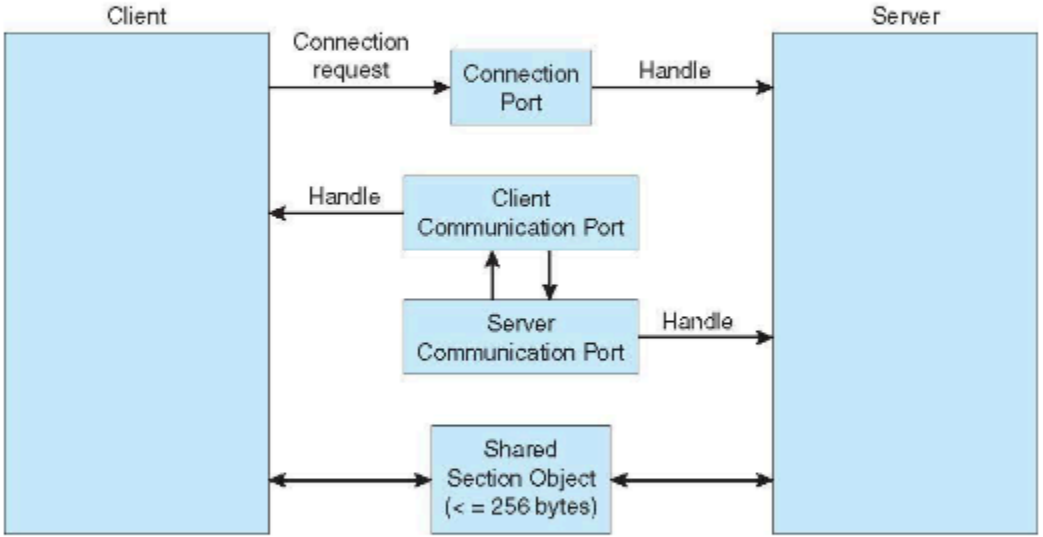
동일한 시스템의 프로세스 간에만 작동통신 채널을 설정하고 유지 관리하기 위해 포트(예: 메일함)를 사용합니다통신은 다음과 같이 작동합니다.

- 클라이언트는 하위 시스템의 연결 포트 개체에 대한 핸들을 엽니다. 클라이언트는 연결 요청을 보냅니다. 서버는 두 개의 개인 통신 포트를 만들고 반환합니다.

그 중 하나에 대한 핸들을 클라이언트로 보냅니다.클라이언트와 서버는 해당 포트 핸들을 사용하여 메시지를 보냅니다.

또는 콜백을 호출하고 응답을 수신 대기합니다.

Windows에서 로컬 프로시저 호출



클라이언트-서버 시스템의 통신

SocketsRemote 프로시저 호출
PipesRemote 메서드 호출
(Java)

소켓

소켓은 통신을 위한 엔드포인트로 정의됩니다

IP 주소와 포트의 연결 – 호스트의 네트워크 서비스를 구별하기 위해 messagepacket의 시작 부분에 포함되는 숫자입니다.

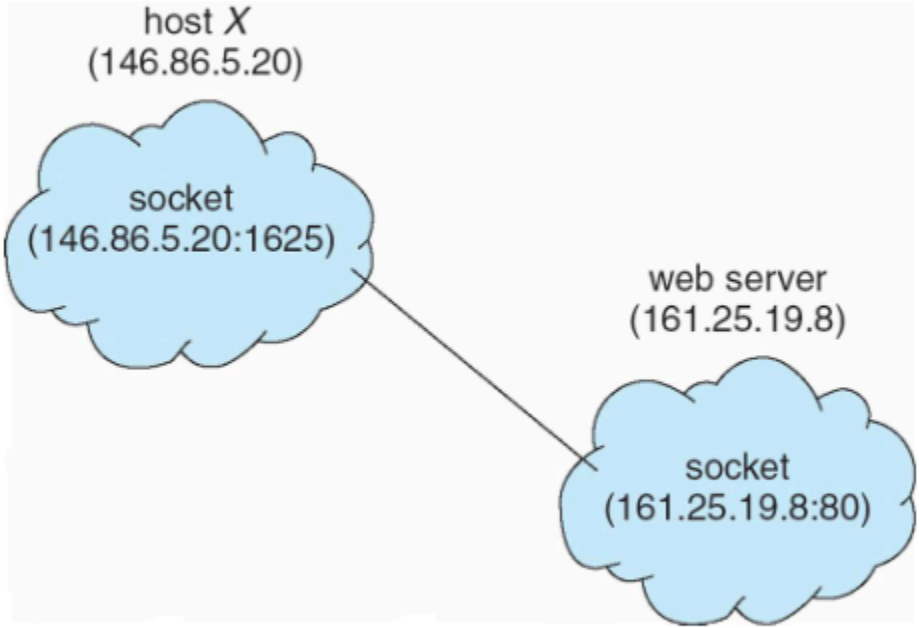
소켓 161.25.19.8:1625는 호스트 161.25.19.8의 포트 1625를 참조합니다.

통신은 한 쌍의 소켓 간에 구성됩니다.

1024 미만의 모든 포트는 잘 알려져 있으며 표준 서비스에 사용됩니다.

특수 IP 주소 127.0.0.1(루프백)은 프로세스가 실행 중인 시스템을 나타냅니다.

소켓 통신


host X
(146.86.5.20)
socket
(146.86.5.20:1625)
web server
(161.25.19.8)
socket
(161.25.19.8:80)

Java의 소켓

세 가지 유형의 소켓

- **연결 지향 (TCP)비연결 (UDP)MulticastSocket 클래스 – 데이터를 여러 수신자에게 보낼 수 있습니다.**

다음 "날짜" 서버를 고려하십시오.

```
import java.net.*;
import java.io.*;

public class DateServer
{
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      /* now listen for connections */
      while (true) {
        Socket client = sock.accept();

        PrintWriter pout = new
         PrintWriter(client.getOutputStream(), true);

        /* write the Date to the socket */
        pout.println(new java.util.Date().toString());

        /* close the socket and resume */
        /* listening for connections */
        client.close();
      }
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

원격 프로시저 호출

원격 프로시저 호출(RPC)은 네트워크로 연결된 시스템의 프로세스 간 프로시저 호출을 추상화합니다

서비스 차별화를 위해 다시 포트 사용스텁 – 서버의 실제 절차를 위한 클라이언트 측 프록시클라이언트 측 스텁은 서버를 찾고 매개 변수를 마샬링합니다.서버 측 스텁은 이 메시지를 수신하고 마샬링된 매개 변수의 압축을 풀고 서버에서 절차를 수행합니다.Windows에서 스텁 코드는 Microsoft로 작성된 사양에서 컴파일됩니다.인터페이스 정의 언어(MIDL))

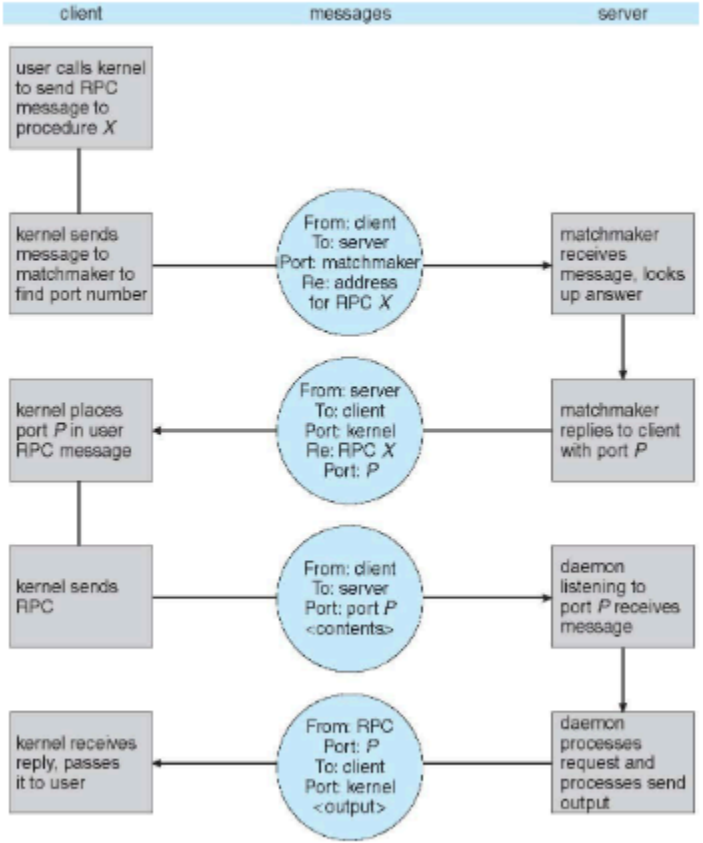
원격 프로시저 호출(계속)

다양한 아키텍처를 고려하기 위해 XDL(External Data Representation) 형식을 통해 처리되는 데이터 표현

빅 엔디안 및 리틀 엔디안원격 통신에는 로컬 통신보다 더 많은 실패 시나리오가 있습니다.

메시지는 최대 한 번이 아닌 정확히 한 번만 전달될 수 있습니다.OS는 일반적으로 클라이언트와 서버를 연결하기 위한 랑데부(또는 매치메이커) 서비스를 제공합니다

RPC 실행


client
messages
server
user calls kernel to send RPC message to procedure X
kernel sends message to matchmaker to find port number
From: client
To: server
Port: matchmaker
Re: address for RPC X
matchmaker receives message, looks up answer
kernel places port P in user RPC message
From: server
To: client
Port: kernel
Re: RPC X
Port: P
matchmaker replies to client with port P
kernel sends RPC
From: client
To: server
Port: port P
<contents>
daemon listening to port P receives message
kernel receives reply, passes it to user
From: RPC
Port: P
To: client
Port: kernel
<output>
daemon processes request and processes send output

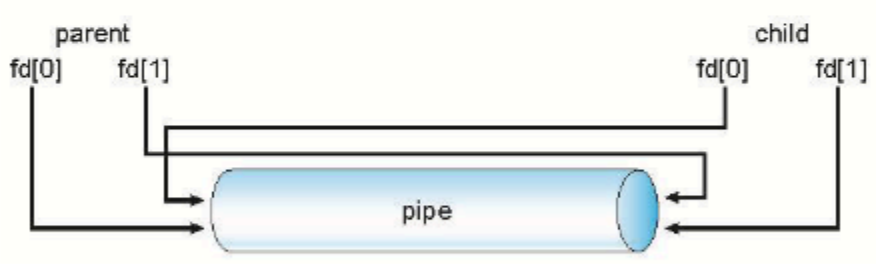
파이프

두 프로세스가 통신할 수 있도록 하는 통로 역할을
합니다.문제:

커뮤니케이션은 단방향입니까, 양방향입니까? 양방향 통신의 경우 반이중입니까 아니면 전이중입니까? 의사소통 과정 사이에 관계(즉, 부모-자식)가 존재해야 합니까? 네트워크를 통해 파이프를 사용할 수 있습니까? 일반 파이프 – 파이프를 생성한 프로세스 외부에서 액세스할 수 없습니다. 일반적으로 부모 프로세스는 파이프를 만들고 이를 사용하여 만든 자식 프로세스와 통신합니다. 명명된 파이프 – 부모-자식 관계 없이 액세스할 수 있습니다.

일반 파이프

일반 파이프는 표준 생산자-소비자 스타일로 통신을 허용합니다.생산자는 한쪽 끝(파이프의 쓰기 끝)에 씁니다.소비자는 다른 쪽 끝(파이프의 읽기 끝)에서 읽습니다.따라서 일반 파이프는 단방향입니다.통신 프로세스 간의 부모-자식 관계가 필요합니다.



Windows는 이러한 익명 파이프를 호출합니다.
교과서에 있는 Unix 및 Windows 코드 샘플 보기

명명된 파이프

명명된 파이프는 일반 파이프보다 강력합니다.통신은 양방향입니다.통신 프로세스 간에 부모-자식 관계가 필요하지 않습니다.여러 프로세스에서 통신을 위해 명명된 파이프를 사용할 수 있습니다.UNIX 및 Windows 시스템 모두에서 제공됩니다.

챕터 3의 끝