3장. 프로세스 (Process)

순천향대학교 컴퓨터공학과
이 상 정

강의 목표 및 내용

- 목표
  - 프로세스의 개념 소개
  - 프로세스의 스케줄링, 생성 및 종료, 통신 등 특성 소개
  - 공유 메모리, 메시지 전달을 사용한 프로세스 간 통신 (IPC)
  - 클라이언트-서버 시스템의 통신
- 내용
  - 프로세스 개념
  - 프로세스 스케줄링
  - 프로세스에 대한 연산
  - 프로세스 간 통신
  - IPC 시스템의 사례
  - 클라이언트 서버 환경에서의 통신

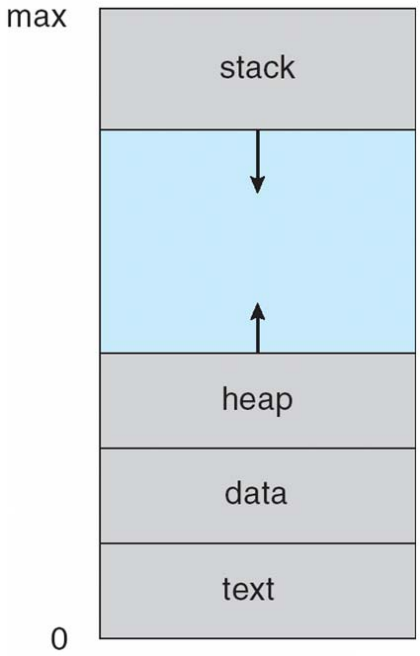
프로세스 개념 (1)

- 운영체제는 다양한 프로그램을 수행
  - 일괄 처리 시스템(batch system)은 작업(job)들을 실행
  - 시분할 시스템(time-shared system)은 사용자 프로그램들 또는 태스크(task)들을 수행
  - 이 책에서는 작업, 태스크와 프로세스란 용어를 거의 호환적으로 사용
- 프로세스는 수행중인 프로그램
- 프로세스는 다음을 포함
  - 텍스트 섹션(text section): 프로그램 코드
  - 프로그램 카운터 (program counter), 레지스터
  - 스택 (stack): 임시 데이터 저장소 (함수 매개변수, 복귀 주소, 지역 변수)
  - 데이터 섹션 (data section): 전역 변수
  - 힙(heap): 실행 중에 동적으로 할당되는 메모리

프로세스 개념 (2)

- 프로그램은 디스크에 저장(실행파일)된 수동적 존재(passive entity)
- 프로세스는 능동적인 존재(active entity)
  - 실행 파일이 메모리에 적재될 때 프로그램은 프로세스가 됨
- 동일한 프로그램이 여러 개의 프로세스가 될 수 있음
  - 다중 인스턴스 (multiple instance)

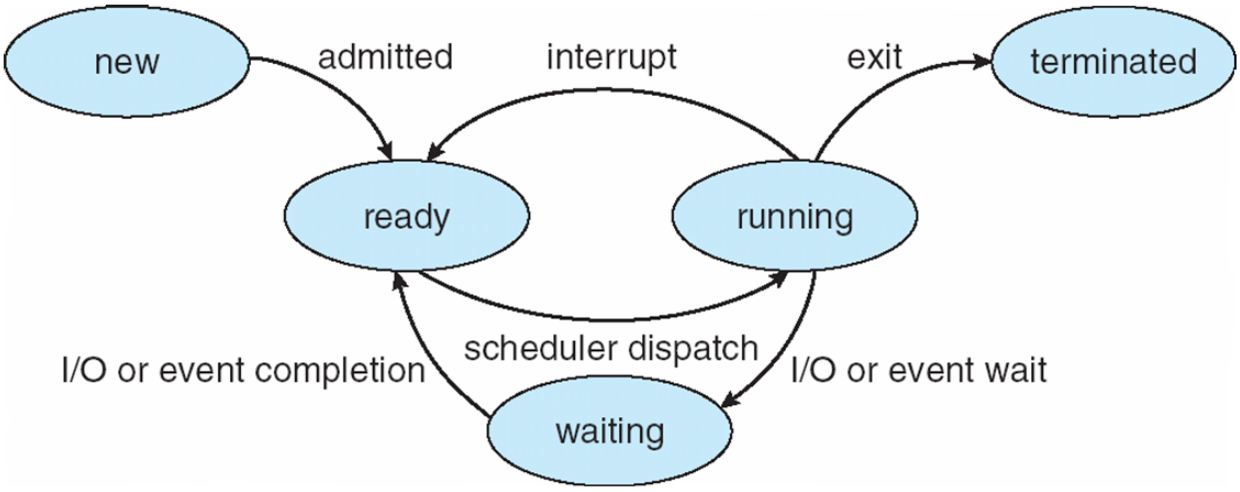
프로세스 메모리 배치



프로세스 상태 (Process State)

- 프로세스는 실행되면서 그 상태가 변함
  - 새로운 (new)
    프로세스가 생성 중임
  - 준비 완료 (ready)
    프로세스가 처리기(processor, CPU)에 할당되기를 기다림
  - 실행 (running)
    명령어들이 실행되고 있음
  - 대기 (waiting)
    프로세스가 어떤 사건(입/출력 완료 또는 신호의 수신 같은)이 일어나기를 기다림
  - 종료 (terminated)
    프로세스의 실행이 종료

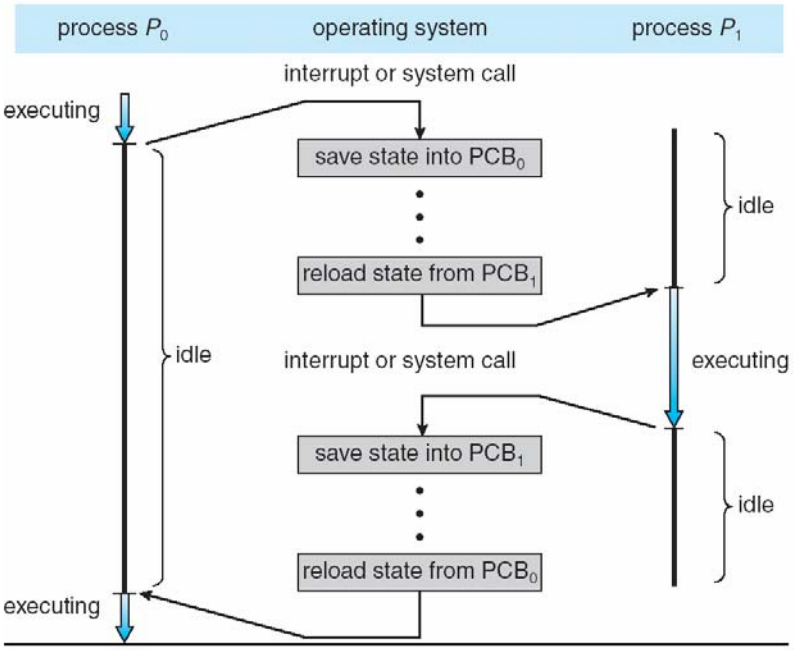
프로세스 상태 다이어그램



프로세스 제어 블록 (Process Control Block, PCB)

- 특정 프로세스와 연관된 여러 정보를 수록 (태스크 제어 블록이라고도 함)
  - 프로세스 상태, 번호
  - 프로그램 카운터
  - CPU 레지스터들
  - CPU 스케줄링 정보
    - 우선순위, 스케줄링 큐 포인터
  - 메모리 관리 정보
    - 프로세스에 할당된 메모리
  - 회계(accounting) 정보
    - CPU 사용량, 경과시간(elapse time)
  - 입/출력 상태 정보
    - 프로세스에 할당된 입출력 장치, 열린 파일 리스트

프로세스 간 CPU 스위치 (문맥 교환)



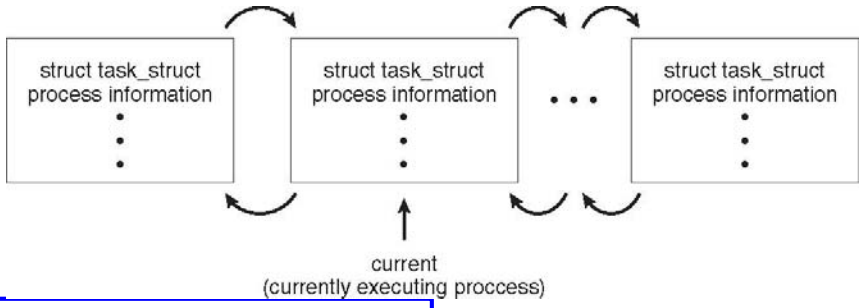
스레드 (Thread)

- 지금까지 프로세스가 하나의 실행 스레드를 갖는 경우를 논의
- 프로세스 당 여러 개의 프로그램 카운터를 갖는 경우를 가정
  - 한 순간에 하나 이상의 일을 수행
    - 워드프로세서에서 키보드 문자를 입력하면서 오자 검사를 하는 예
  - 다중 스레드(multiple thread, multithread)
- PCB에 여러 개의 프로그램 카운터 등 스레드에 관한 정보 저장
- 다음 장에서 소개

리눅스에서 프로세스 표현

- C 언어의 struct task_struct로 표현
  - 필드 예

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

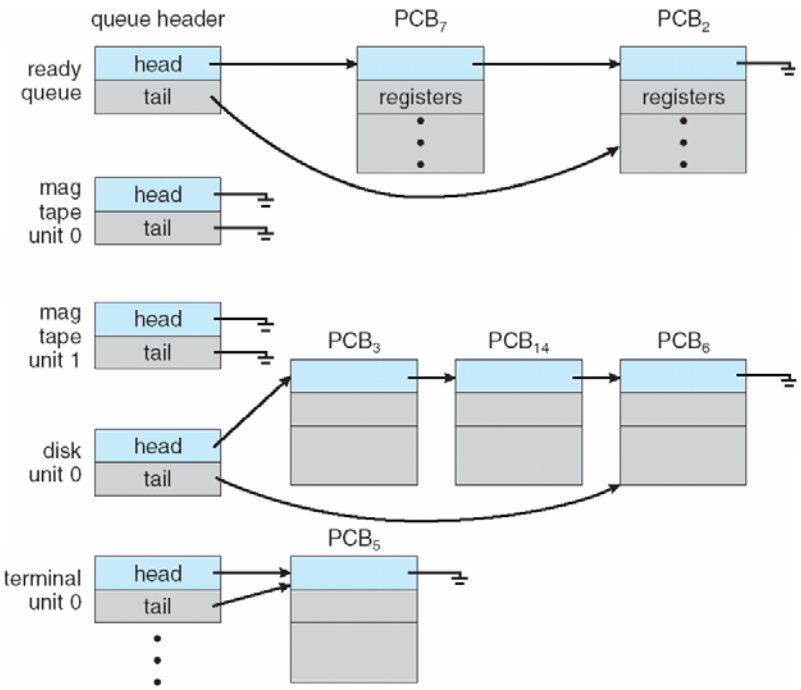


프로세스 스케줄링 (Process Scheduling)

- 다중 프로그래밍은 CPU의 이용을 최대화하고, 시분할은 프로세스들 간의 빠른 전환이 되어야 함
- 프로세스 스케줄러는 CPU에서 수행 가능한 여러 프로세스들 중에서 하나의 프로세스를 선택

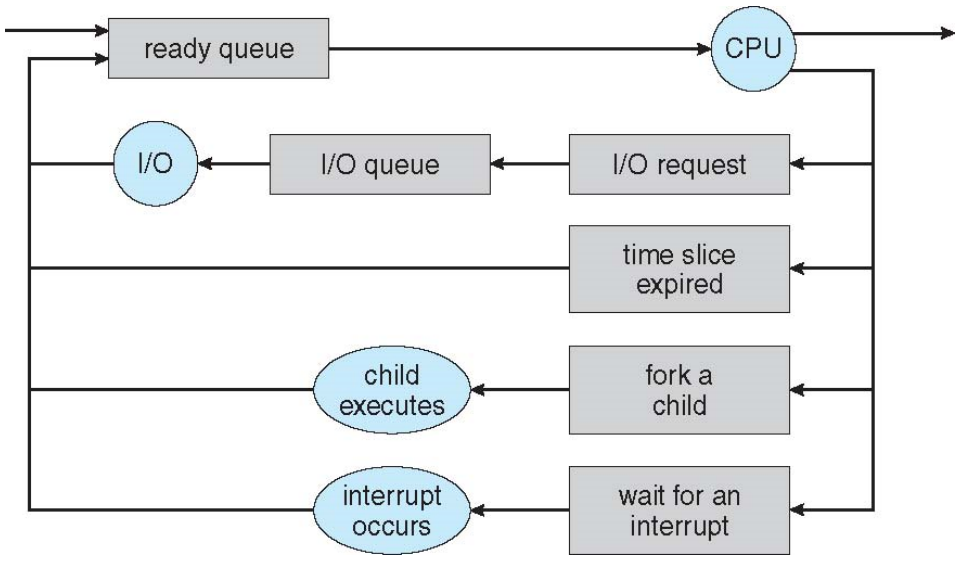
- 스케줄링 큐 (scheduling queue)
  - 작업 큐 (job queue)
    시스템 안의 모든 프로세스들로 구성
  - 준비 큐(ready queue)
    주 메모리에 존재하며, 준비 완료 상태에서 실행을 대기하는 프로세스들로 구성
  - 장치 대기 큐 (device queue)
    특정 입/출력장치를 대기하는 프로세스들의 리스트들로 구성
  - 프로세스는 일생 동안에 다양한 스케줄링 큐들 사이를 이동

준비 큐와 다양한 입/출력 장치 대기 큐



프로세스 스케줄링 표현
- 큐잉 다이어그램 (Queuing Diagram)

- 큐잉 다이어그램은 큐, 자원, 흐름을 표시



스케줄러 (Scheduler) (1)

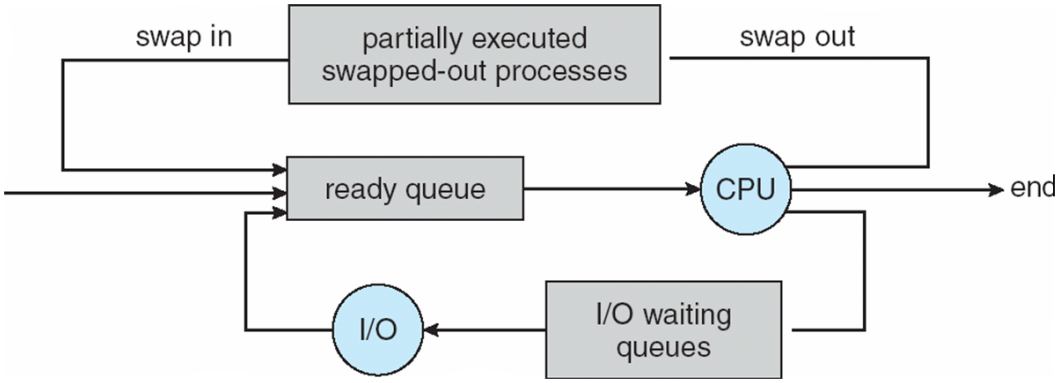
- 단기 스케줄러(또는 CPU 스케줄러)
  - (메모리에 있는) 실행 준비가 완료되어 준비 큐에 있는 프로세스들 중에서 선택하여, 이들 중 하나에게 CPU를 할당
  - 단기 스케줄러는 자주 (milliseconds) 수행되므로 빨라야 함
- 장기 스케줄러(또는 작업 스케줄러)
  - (디스크 상의) 프로세스를 선택하여 준비 큐로 저장
  - 실행하기 위해 메모리로 적재
  - 가끔 실행 됨 (seconds, minutes)
  - 장기 스케줄러는 다중 프로그래밍의 정도(multiprogramming granularity, 메모리에 있는 프로세스들의 수)를 제어

스케줄러 (2)

- 대부분의 프로세스들은 입/출력 중심 또는 CPU 중심
  - 입/출력 중심 프로세스 (I/O-bound process)
    연산보다 입/출력 수행에 더 많은 시간을 소요하는 프로세스
  - CPU 중심 프로세스 (CPU-bound process)
    연산에 시간을 더 소요하여, 입/출력 요청을 드물게 발생시키는 프로세스
- 장기 스케줄러는 입출력 중심과 CPU 중심 프로세스들의 적절한 프로세스 혼합(mix)를 선택하는 것이 중요
- 장기 스케줄러의 기능이 없거나 최소화된 운영체제
  - UNIX나 윈도우즈와 같은 시분할 시스템

중기 스케줄러 (Medium-term Scheduler)

- 시분할 시스템과 같은 일부 운영체제들은 추가로 중간 수준의 스케줄링을 도입
  - 메모리에서 (CPU를 위해 적극적으로 경쟁하는) 프로세스들을 제거하여 다중 프로그래밍의 정도를 완화
  - 차후에 다시 프로세스를 메모리로 불러와서 중단되었던 지점에서 부터 실행을 재개 => 스와핑(swapping)



모바일 시스템에서의 다중태스킹

- 초기 시스템은 모바일 장치의 제약 조건 때문에 다중태스킹(multi-tasking)을 지원하지 않았음
- 애플 iOS 4를 시작으로 제한된 형태의 다중 태스킹 허용
  - 단일 전경(foreground) 프로세스
    - 화면에 보이면서 실행 중인 응용
  - 다수의 백그라운드(background) 프로세스
    - 메모리에 남아 있고 실행 중이지만 화면에 보이지 않는 응용들
    - 단일 작업을 수행하는 짧은 태스크 (예,다운로드), 사건의 공지(event notification)를 수신하는 응용 (예, 메일 수신), 실행 시간이 긴 응용 (예, 오디오 재생)
- 안드로이드는 백그라운드 프로세스 응용 제한이 없음
  - 백그라운드 응용은 태스크 수행을 위해 서비스를 사용
  - 백그라운드 응용이 보류되어도 서비스는 실행
  - 서비스는 사용자 인터페이스를 가지고 있지 않고 적은 메모리를 사용

문맥 교환 (Context Switch)

- CPU를 다른 프로세스로 교환하려면 현재 프로세스의 상태를 보관하고 새로운 프로세스의 보관된 상태를 복구하는 작업이 필요
  - 이 작업을 문맥 교환(context switch)이라고 함
- 프로세스의 문맥(context)는 PCB에 저장된 내용을 표현
- 문맥 교환이 진행될 동안 시스템은 아무런 유용한 일을 못하기 때문에 문맥 교환 시간은 순수한 오버헤드(overhead)
  - 메모리의 속도와 반드시 복사되어야 하는 레지스터의 수 등에 처리 속도가 좌우되므로, 기계마다 다름
  - 전형적인 속도는 수 마이크로초까지 분포
  - 문맥 교환 시간은 하드웨어의 지원에 크게 좌우

프로세스 생성 (Process Creation) (1)

- 프로세스는 실행 도중에, 프로세스 생성 시스템 콜을 통해서 여러 개의 새로운 프로세스들을 생성
  - 생성하는 프로세스를 부모 프로세스(parent process), 새로운 프로세스는 자식 프로세스(children process)
  - 생성 결과 프로세스의 트리를 형성
  - 프로세스 식별자 (pid, process identifier)를 사용하여 구분
  - 생성된 프로세스는 운영체제로부터 직접 자원(resource)을 얻거나, 부모 프로세스 자원의 부분 집합을 사용
- 프로세스 간 자원 공유(resource sharing) 방식
  - 부모 자식 프로세스 간 모든 자원 공유
  - 자식 프로세스가 부모 프로세스 자원의 일부분만을 공유
  - 부모와 자식 간 자원을 공유하지 않음

프로세스 생성 (2)

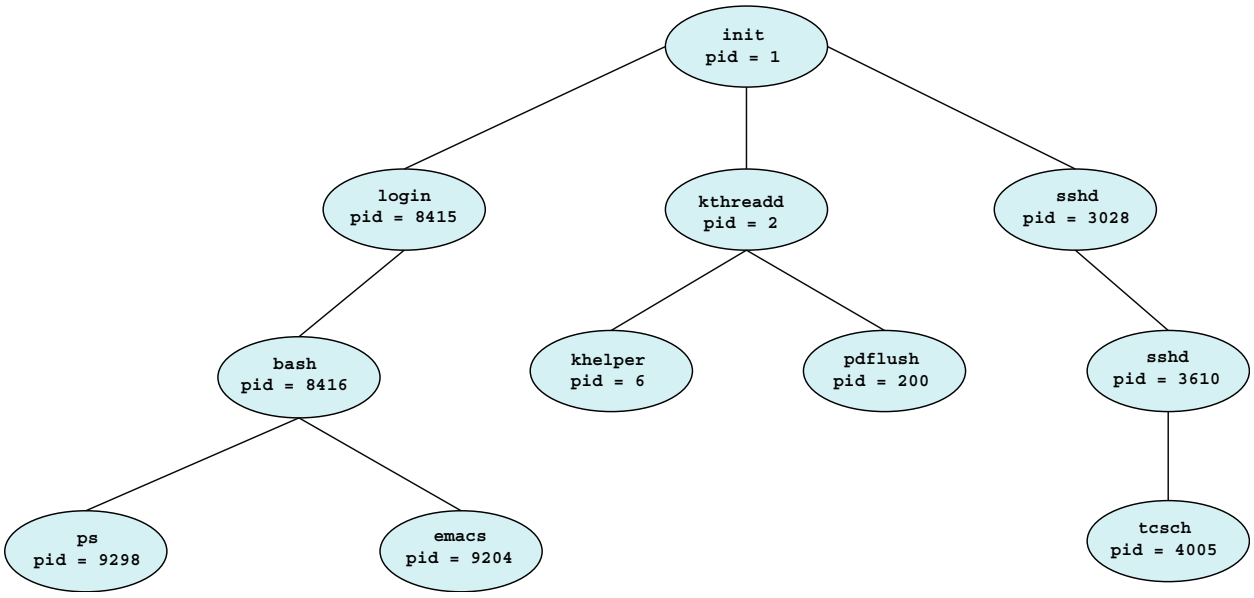
❑ 실행 (execution) 방식

- 부모가 계속해서 자식과 병행 수행(concurrent execution)
- 부모가 모든 자식 또는 일부 자식이 끝날 때까지 기다림

❑ 주소 공간 (address space)

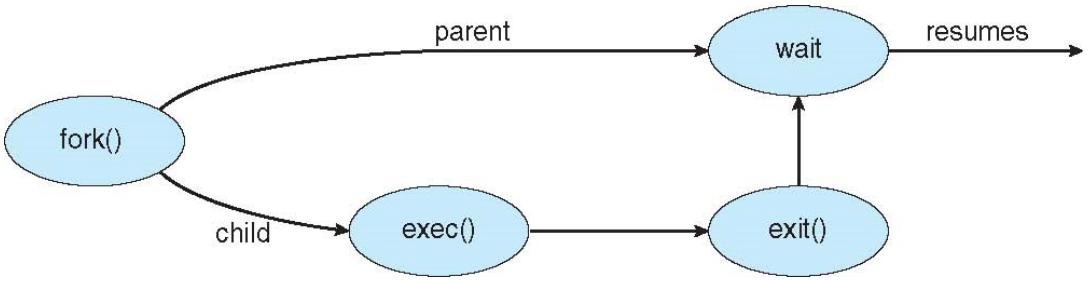
- 자식 프로세스는 부모 프로세스의 복사본 (자식 프로세스는 부모와 똑같은 프로그램과 데이터를 가짐)
- 자식 프로세스가 자신에게 적재될 새로운 프로그램을 가짐

리눅스의 프로세스 트리 예



프로세스 생성 (3)

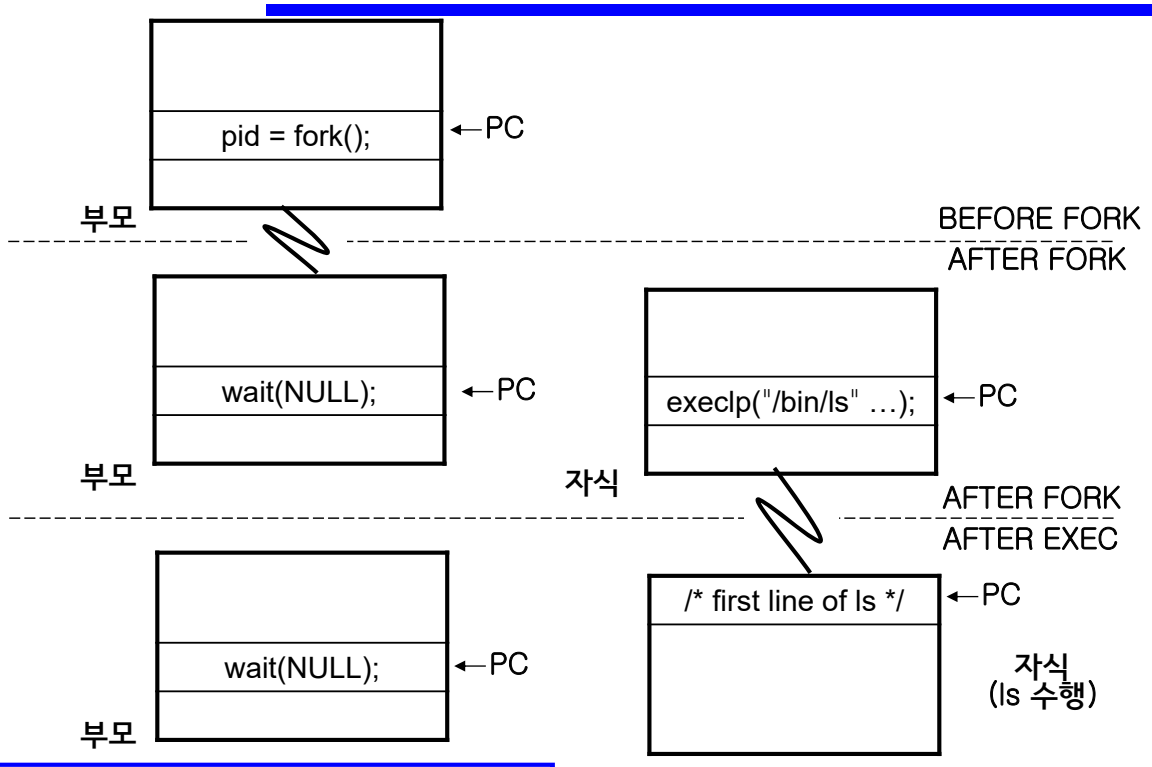
- 새로운 프로세스는 fork() 시스템 콜로 생성
  - 새로운 프로세스는 원래 프로세스의 주소 공간의 복사본으로 구성
  - 부모 프로세스가 쉽게 자식 프로세스와 통신
- fork() 시스템 콜 후 자신의 메모리 공간을 새로운 프로그램으로 바꾸기 위해서 exec() 시스템 콜을 사용
  - exec() 시스템 콜은 이진 파일을 메모리로 적재(load)하고 실행



새 프로세스를 fork하는 C 프로그램 (1)

```
#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;
}
```

새 프로세스를 fork하는 C 프로그램 (2)



프로세스 종료 (Process Termination) (1)

- 프로세스가 마지막 문장의 실행을 끝내고, exit() 시스템 콜을 사용하여 운영체제에게 자신의 삭제를 요청하면 종료
  - 프로세스는 자신의 부모 프로세스에게 (wait() 시스템 콜을 통해) 상태 값(통상 정수값)을 반환
  - 물리 메모리와 가상 메모리,열린 파일, 입/출력 버퍼를 포함한 프로세스의 모든 자원이 운영체제로 반납

- 부모는 다음과 같이 여러 가지 이유로 인하여 자식들 중 하나의 실행을 종료 (abort())
  - 자식이 자신에게 할당된 자원을 초과하여 사용할 때
  - 자식에게 할당된 태스크(task)가 더 이상 필요 없을 때
  - 부모가 종료된 후 자식들의 실행이 계속되는 것을 허용하지 않는 경우

프로세스 종료 (2)

- 부모가 종료(exit)한 후에 자식이 수행을 계속하는 것을 허용하지 않는 운영체제도 있음
  - 모든 자식 프로세스들도 종료, 연속적 종료(cascading termination)
- 부모 프로세스는 wait() 시스템 콜을 사용하여 자식 프로세스의 종료를 기다림
  - wait() 시스템 콜은 상태 정보와 종료된 프로세스의 pid를 반환
    pid = wait(&status);
  - 부모 프로세스가 wait() 호출하지 않은 상태에서 종료되는 자식 프로세스는 좀비(zombie)
  - 부모 프로세스가 wait() 호출 없이 종료하면, 이 때의 자식 프로세스는 고아(orphan)

좀비 프로세스 예 (1)

```
// zombie.c

#include <stdio.h>
#include <stdlib.h>

int main ()
{
   int pid;

   pid = fork ();     // 새 프로세스 생성

    if (pid > 0) {     //  부모 프로세스
        while (1)      // 무한 루프, 자식 기다리지 않음 (wait() 없음)
            sleep (1000);
    }
    else                    // 자식 프로세스
            exit (0);       // 종료, 그러나 부모는 기다리지 않음

    return 0;
}
```

좀비 프로세스 예 (2)

```
lee@leeVB:~/os$ gcc -o zombie zombie.c
lee@leeVB:~/os$
lee@leeVB:~/os$ ./zombie &
[1] 3255
lee@leeVB:~/os$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1982  1969  0  80   0 -  2135 wait   pts/2    00:00:00 bash
0 S  1000  3255  1982  0  80   0 -   496 hrtime pts/2    00:00:00 zombie
1 Z  1000  3256  3255  0  80   0 -     0 exit   pts/2    00:00:00 zombie <defunct>
0 R  1000  3257  1982  0  80   0 -  1477 -      pts/2    00:00:00 ps
lee@leeVB:~/os$
lee@leeVB:~/os$ kill -9 3255
lee@leeVB:~/os$
[1]+  죽었음                  ./zombie
lee@leeVB:~/os$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1982  1969  0  80   0 -  2135 wait   pts/2    00:00:00 bash
0 R  1000  3258  1982  0  80   0 -  1477 -      pts/2    00:00:00 ps
lee@leeVB:~/os$
```


멀티프로세스 구조 - 크롬(Chrome) 브라우저

- 기존에는 많은 웹 브라우저들이 단일 프로세스로 실행되었음 (일부는 여전히 단일 프로세스)
  - 방문한 웹 사이트에 문제가 있으면 전체 브라우저가 깨짐
- 구글 크롬 브라우저는 멀티프로세스로 (3가지 형태의 프로세스) 동작
  - 브라우저 (Browser) 프로세스는 사용자 인터페이스, 디스크, 네트워크 입출력을 관리
  - 렌더러(Renderer) 프로세스는 웹페이지를 표시하고, HTML,Javascript를 다룸. 각 웹 사이트가 오픈될 때 새로운 렌더러가 생성
    - 샌드박스(sandbox) 기능을 사용하여 시스템 액세스 정도를 제한하여 보안을 강화
  - 플러그인(Plug-in) 프로세스는 플러그인된 특정 콘텐츠를 위해 목록에 표시

fork 실습과제

- p.132 그림 3.8의 프로그램을 일부 수정한 프로그램(fork.c)이다. 다음과 같이 두 개의 터미널에서 실행하고 결과를 분석하여라.
  - 터미널1에서 소스 작성하고 실행 시작 후 문자 입력 전 대기
  - 터미널2를 생성하고 다음을 실행하고 출력 분석
    $ ps -al
  - 터미널 1에서 문자 입력하고 실행 결과 분석
  - 컴파일 시 warning error 제거하도록 프로그램 보완

- VirtualBox 과제 캡처 시 주의 사항
  - 사용자 셀 프롬트에 자기 이름 표시
  - 프롬트 변경 예
    $ export PS1="(이름)$ "


```
#include <stdio.h>

int main( )
{
  int pid;   /* pid_t pid; */
  // 여기에 자신의 학번, 이름 출력
  /* 새 프로세스를 생성한다(fork) */
  pid = fork();

  /* 입력 대기 */
  printf("Input any character to continue\n");
  getchar();

  if (pid < 0) {     /* 오류가 발생했음 */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if (pid == 0) {      /* 자식 프로세스 */
    execlp("/bin/ls", "ls", NULL);
  }
  else {       /* 부모 프로세스 */
    /* 부모가 자식이 완료되기를 기다림 */
    wait(NULL);
    printf("Child Complete");
    exit(0);
  }
}
```

```
lee@leeVB: ~/os_test
lee@leeVB:~/os_test$ cc -o fork fork.c
fork.c: In function ‘main’:
fork.c:16:7: warning: incompatible implicit declaration of built-in function ‘exit’ [ena
bled by default]
fork.c:19:7: warning: incompatible implicit declaration of built-in function ‘execlp’ [e
nabled by default]
fork.c:25:7: warning: incompatible implicit declaration of built-in function ‘exit’ [ena
bled by default]
lee@leeVB:~/os_test$
lee@leeVB:~/os_test$ ./fork
Input any character to continue
Input any character to continue
```

```
lee@leeVB: ~/os_test
lee@leeVB:~/os_test$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2847  1885  0  80   0 -   503 n_tty_ pts/1    00:00:00 fork
1 S  1000  2848  2847  0  80   0 -   503 n_tty_ pts/1    00:00:00 fork
0 R  1000  2849  2589  0  80   0 -  1482 -      pts/3    00:00:00 ps
lee@leeVB:~/os_test$
```



```
lee@leeVB: ~/os_test
lee@leeVB:~/os_test$ cc -o fork fork.c
fork.c: In function ‘main’:
fork.c:16:7: warning: incompatible implicit declaration of built-in function ‘exit’ [ena
bled by default]
fork.c:19:7: warning: incompatible implicit declaration of built-in function ‘execlp’ [e
nabled by default]
fork.c:25:7: warning: incompatible implicit declaration of built-in function ‘exit’ [ena
bled by default]
lee@leeVB:~/os_test$
lee@leeVB:~/os_test$ ./fork
Input any character to continue
Input any character to continue

h
fork    fork.c~    shm-posix.c   shm-posix2    shm-posix2.c~
fork.c  shm-posix  shm-posix.c~  shm-posix2.c  shm1.c
Child Completelee@leeVB:~/os_test$
```

```
lee@leeVB: ~/os_test
lee@leeVB:~/os_test$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2847  1885  0  80   0 -   503 n_tty_ pts/1    00:00:00 fork
1 S  1000  2848  2847  0  80   0 -   503 n_tty_ pts/1    00:00:00 fork
0 R  1000  2849  2589  0  80   0 -  1482 -      pts/3    00:00:00 ps
lee@leeVB:~/os_test$
lee@leeVB:~/os_test$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 R  1000  2850  2589  0  80   0 -  1474 -      pts/3    00:00:00 ps
lee@leeVB:~/os_test$
```

fork 연습문제

- 연습문제 3.1
  - 그림 3.30 프로그램을 앞의 실습과 같이 프로그램을 수정하여 실행 후 실행 결과 분석

프로세스간 통신 (1) (Interprocess Communication)

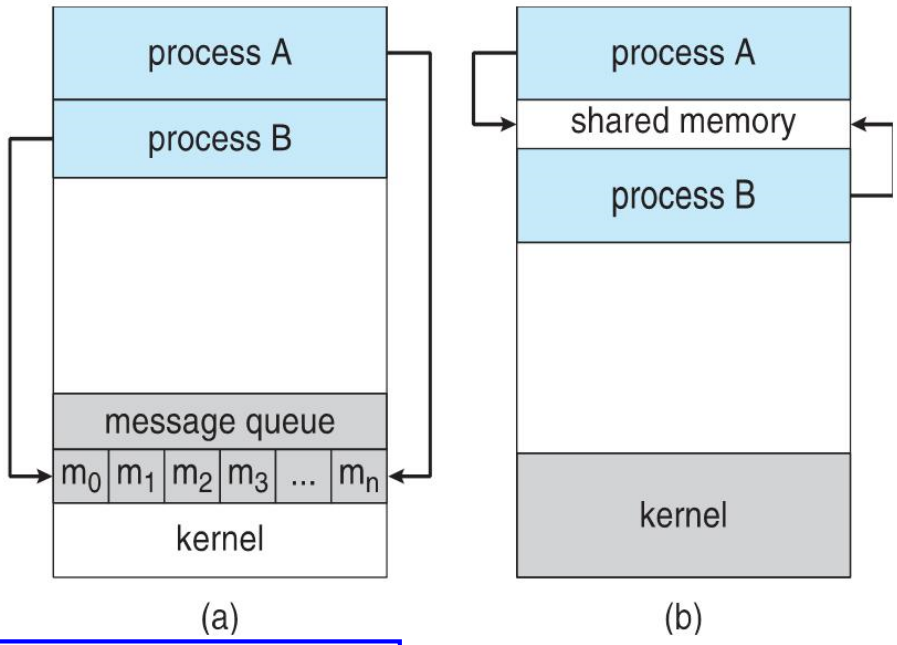
- 운영체제 내에서 수행되는 병행 프로세스들은 독립적(independent)이거나 또는 협력적인(cooperative) 프로세스
  - 독립적 프로세스는 실행 중인 다른 프로세스들에게 영향을 주거나 받지 않음
  - 협력적 프로세스는 실행중인 다른 프로세스들에게 영향을 주거나 받음
- 프로세스 협력을 허용하는 환경을 제공하는 이유
  - 정보 공유 (information sharing)
  - 계산 가속화 (computation speedup)
  - 모듈성 (modularity)

프로세스간 통신 (2)

- 협력적 프로세스들은 데이터와 정보를 교환할 수 있는 프로세스간 통신(InterProcess Communication, IPC) 기법 필요
- 프로세스간 통신의 두 가지 모델
  - 공유 메모리(shared memory)
    - 협력 프로세스들에 의해 공유되는 메모리의 영역이 구축
    - 프로세스들은 공유 영역에 데이터를 읽고 쓰고 함으로써 정보를 교환
    - 하나의 컴퓨터 안에서 통신을 할 때 메모리 속도로 접근 가능하기 때문에 최대 속도와 편의성을 제공
  - 메시지 전달(message passing)
    - 협력 프로세스들 사이에 교환되는 메시지를 통하여 통신
    - 충돌을 회피할 필요가 없기 때문에 적은 양의 데이터를 교환하는데 유용
    - 공유 메모리 모델보다 구현이 용이

프로세스 간 통신 모델

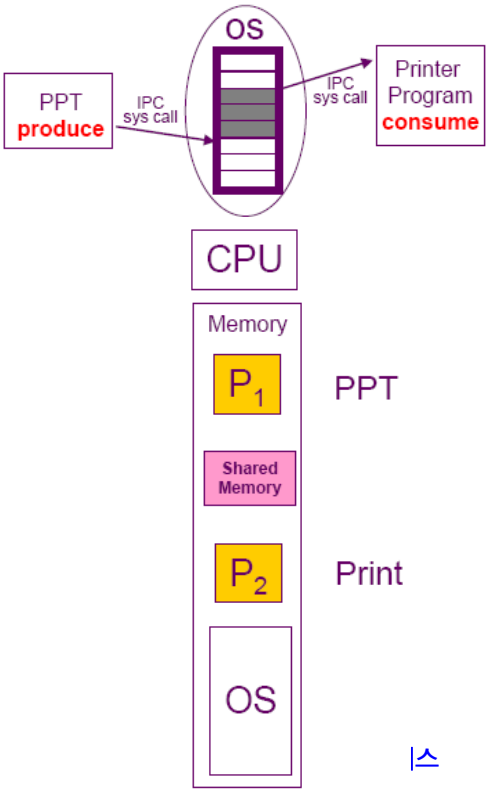
(a) Message passing. (b) shared memory.



공유 메모리 - 생산자/소비자 문제

생산자-소비자 문제 (Producer-Consumer Problem)

- 협력하는 프로세스의 일반적인 패러다임
- 생산자 프로세스는 정보를 생산하고 소비자 프로세스는 정보를 소비
- 생산자-소비자 문제의 하나의 해결책은 공유 메모리를 사용
- 두 가지 유형의 버퍼가 사용
  - 무한 버퍼(unbounded buffer)의 생산자와 소비자 문제에서는 버퍼의 크기에 실질적인 한계가 없음
  - 유한 버퍼(bounded buffer)는 버퍼의 크기가 고정되어 있다고 가정

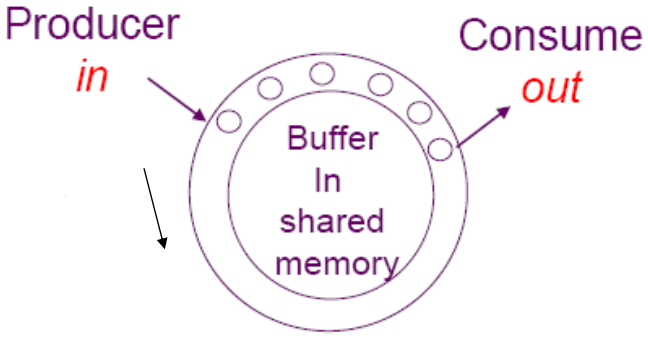


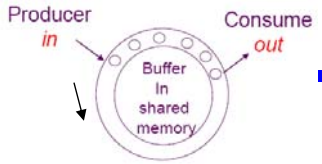
유한 버퍼 - 공유 메모리 해법

공유 데이터

- 공유 버퍼는 두 개의 논리 포인터 in과 out을 갖는 원형 배열로 구현
- in == out 일 때 버퍼는 비어 있고, ((in + 1) % BUFFER_SIZE)==out이면 버퍼는 가득 차 있음을 표시
- 최대 BUFFER_SIZE - 1까지만을 버퍼에 데이터 수용

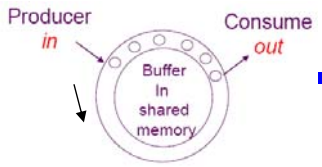
```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

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





유한 버퍼 - 생산자 프로세스

```
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;
}
```

유한 버퍼 - 소비자 프로세스

```
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 */
}
```

공유 메모리 - 요약

- 서로 통신하고자 하는 프로세스들 간에 메모리 영역 공유
- 운영체제가 아닌 사용자 프로세스들의 제어 하에 통신
- 프로세스들이 병행 실행될 때 공유 버퍼(메모리)에 접근의 동기화가 주요 문제 (6, 7장 소개)

메시지 전달 시스템 (Message-Passing System)

- 메시지 전달 방식은 동일한 주소 공간을 공유하지 않고도 프로세스들이 통신을 하고 동작을 동기화
  - 서로 다른 컴퓨터의 프로세스들 간에 통신하는 분산 환경에 적합
  - 메시지 전달 시스템의 연산
    - send(message)
    - receive(message)
    - 프로세스가 보낸 메시지는 고정 길이 또는 가변 길이
  - 만약 프로세스 P와 Q가 통신을 원하면
    - 통신 연결(communication link)이 설정
    - send()/receive() 연산을 사용하여 메시지 교환
  - 통신 연결 구현
    - 물리적인 구현(공유 메모리, 하드웨어 버스, 네트워크 등)
    - 논리적 구현
      - 직접 또는 간접 통신

메시지 전달 - 직접 통신 (Direct Communication)

- 각 프로세스는 통신의 수신자 또는 송신자의 이름을 명시
  - send(P, message) : 프로세스 P에게 메시지를 전송
  - receive(Q, message) : 프로세스 Q로부터 메시지를 수신
- 통신 연결의 특성
  - 통신을 원하는 각 프로세스의 쌍들 사이에 연결이 자동적으로 구축
  - 연결은 정확히 두 프로세스들 사이에만 연관
  - 통신하는 프로세스들의 각 쌍 사이에는 정확하게 하나의 연결이 존재
- 직접 통신 문제점
  - 프로세스의 이름을 바꾸면 모든 다른 프로세스 정의를 검사할 필요
  - 옛 이름들에 대한 모든 참조를 반드시 찾아서 새로운 이름으로 변경
  - 간접 통신으로 해결

메시지 전달 - 간접 통신 (1) (Indirect Communication)

- 메시지들은 메일박스(mailbox) 또는 포트(port)로부터 송수신
  - 메일박스는 고유의 id를 가짐
  - 두 프로세스들이 공유 메일박스를 가질 때만 이들 프로세스가 통신
  - send(A, message) : 메시지를 메일박스 A로 송신
  - receive(A, message) : 메시지를 메일박스 A로부터 수신
- 통신 연결의 특성
  - 한 쌍의 프로세스들 사이의 연결은 이들 프로세스가 공유 메일박스를 가질 때만 구축
  - 연결은 두 개 이상의 프로세스들과 연관
  - 통신하고 있는 각 프로세스들 사이에는 다수의 서로 다른 연결이 존재

메시지 전달 -간접 통신 (2)

- 간접 통신 동작
  - 새로운 메일박스를 생성
  - 메일박스를 통해 메시지를 송신하고 수신
  - 메일박스를 삭제
- 메일박스 공유
  - 문제점
    - 프로세스 P1, P2, P3가 모두 메일박스 A를 공유
    - 프로세스 P1은 메시지를 A에 송신하고, P2와 P3는 각각 A로부터 수신
    - 어느 프로세스가 P1이 보낸 메시지를 수신하는가?
  - 해법
    - 하나의 링크는 최대 두 개의 프로세스와 연관되도록 허용
    - 한 순간에 최대로 하나의 프로세스가 receive() 연산을 실행하도록 허용
    - 어느 프로세스가 메시지를 수신할 것인지 시스템이 임의로 선택하고, 시스템은 송신자에게 수신자가 누구인지 알려줌

메시지 전달 - 동기화 (Synchronization) (1)

- 메시지 전달은 봉쇄형(blocking)이거나 비봉쇄형(nonblocking) (또는 동기식 또는 비동기식이라고도 함)
- 봉쇄형 송수신
  - 봉쇄형 보내기: 송신하는 프로세스는 메시지가 수신 프로세스 또는 메일박스에 의해 수신될 때까지 봉쇄(blocking)
  - 봉쇄형 받기: 메시지가 이용 가능할 때까지 수신 프로세스가 봉쇄
- 비봉쇄형 송수신
  - 비봉쇄형 보내기: 송신하는 프로세스가 메시지를 보내고 작업을 재시작
  - 비봉쇄형 받기: 수신하는 프로세스가 유효한 메시지 또는 널(null)을 받음
- send(), receive()의 동기화 조합
  - 각각 다른 조합도 가능
  - send() receive(), 모두 봉쇄형인 경우 송수신자 간에 랑데부(rendezvous)
    - 생산자, 소비자 동기화 문제는 해결
    - 생산자는 수신자에게 전달될 때까지 기다림

메시지 전달 - 생산자/소비자 랑데부

```
message next_produced;
while (true) {
    /* produce an item in next produced */
send(next_produced);
}
```

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

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

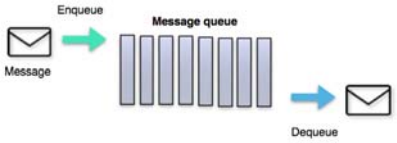

메시지 전달 - 버퍼링 (Buffering)

- 메시지 전달 통신이 직접적이든 간접적이든 간에, 통신하는 프로세스들에 의해 교환되는 메시지는 임시 큐에 저장

- 큐를 구현하는 방식
  - 무용량 (zero capacity)
    - 큐의 최대 길이가 0
    - 송신자는 수신자가 메시지를 수신할 때까지 기다림
  - 유한 용량 (bounded capacity)
    - 큐는 유한한 길이 n을 가짐
    - 링크가 만원이면, 송신자는 큐 안에 공간이 이용 가능할 때까지 봉쇄
  - 무한 용량(unbounded capacity)
    - 큐는 잠재적으로 무한한 길이를 가짐
    - 송신자는 결코 봉쇄되지 않음

참고 - 메시징 시스템 (Messaging System)

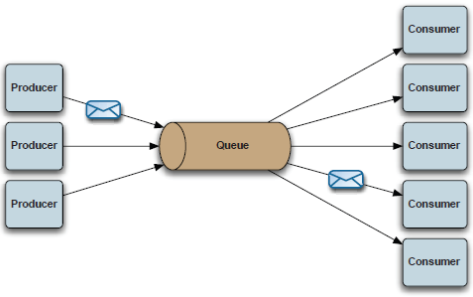
- 대규모의 빅데이터를 수집하고 분석하려면 메시징 시스템 (messaging system)이 필요



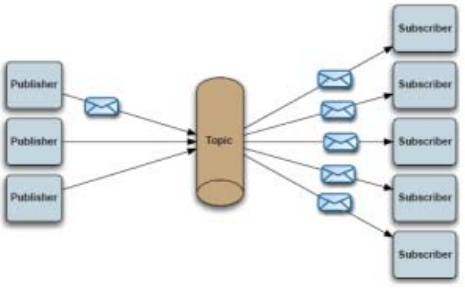
- 메시징 시스템은 한 응용의 데이터를 다른 응용으로 전송
  - 응용에서는 데이터에만 초점을 두고 어떻게 데이터가 전송되고 공유되는지 알 필요 없음
  - 분산 메시징(distributed messaging)은 신뢰적인 메시지 큐잉 (reliable message queuing)에 기반
    - 클라이언트 응용들과 메시징 시스템 사이에서 메시지들이 비동기적으로 큐잉
  - 두 종류의 메시지 모델
    - 점-대-점 (point-to point) 모델
    - 배포-구독(Publish-subscribe) 모델

참고 - 메시징 모델

- 점-대-점(point-to-point) 모델
  - 큐에 저장된 메시지들은 하나 이상의 소비자(consumer)가 소비
  - 특정 메시지는 특정 소비자만 소비

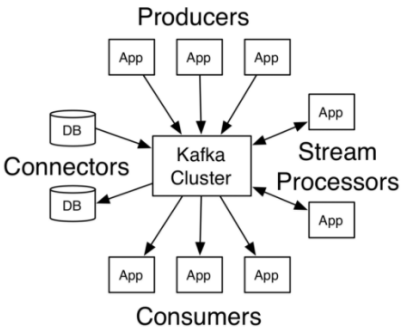


- 배포-구독(publish-subscribe) 모델
  - 메시지가 토픽(topic)으로 저장
  - 여러 소비자들이 하나 이상의 토픽을 구독하고 해당 토픽의 모든 메시지들을 소비
    - 메시지 생산자를 배포자(publisher), 소비자를 구독자(subscriber)라 함



참고 - 아파치 카프카

- 아파치 카프카(Apache Kafka)
  - 분산 배포-구독 메시징 시스템 (distributed publish-subcribe messaging system)
  - 레코드들의 스트림들을 배포하고 구독
  - 고장-감내(fault-tolerant) 방식으로 스트림들을 저장
  - 레코드들의 스트림 발생 시점에서 처리
  - 하나 이상의 서버들로 구성된 클러스터에서 동작
  - Fortune 500대 기업 중 1/3이 카프카 사용
  - https://kafka.apache.org



POSIX 공유 메모리 함수 (1)

- 공유 메모리 객체 생성하여 파일기술자(file descriptor)를 반환, 실패 시 -1 리턴

```
#include <sys/mman.h>
#include <fcntl.h>                    // O_ 플래그 정의
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

  - 인자1은 공유메모리 객체의 이름, name
  - 인자2는 객체가 존재하지 않으면 생성(O_CREAT)하고, 읽기와 쓰기 가능 상태(O_RDWR) 오픈 됨을 지정,
  - 인자3은 객체의 사용권한 모드, 0666

- 공유 메모리 객체의 크기를 지정 (바이트 단위), 성공 시 0 실패 시 -1 리턴

```
#include <unistd.h>
ftruncate(shm_fd, 4096);
```

  - 객체의 크기를 4096 바이트로 설정

POSIX 공유 메모리 함수 (2)

- 공유 메모리 객체를 메모리에 사상하여 접근할 포인터(주소) 반환

```
#include <sys/mman.h>
ptr = mmap(NULL, 4096, PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

  - 인자1은 시작주소를 제안하는 인자이나 일반적으로 NULL(0)로 지정
  - 인자2는 사상이 되는 메모리 영역의 크기, 4096
  - 인자3은 메모리 보호, PROT_WRITE는 쓰기가 가능한 페이지 표시
  - 인자4는 사상의 유형 표시, MAP_SHARED는 프로세스들 간 공유를 표시
  - 인자5는 사상할 객체의 파일기술자 , shm_fd
  - 인자6은 사상할 메모리 영역의 시작에서부터의 오프셋

- 공유 메모리 객체 제거, 성공 시 0 실패 시 -1 리턴

```
#include <sys/mman.h>
shm_unlink(name)
```


POSIX 공유 메모리 예 - shprod.c

```
// shprod.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>

int main()
{
        const int SIZE = 4096;
        const char *name = "OS";
        const char *message0= "Hello ";
        const char *message1= "Soonchunhyang ";
        const char *message2= "Computer Engineering and Science!\n";

        int shm_fd;
        void *ptr;

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

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

    /* now map the shared memory segment in the address space of the process */
    ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
            printf("Map failed\n");
            return -1;
    }

    /**
     * Now write to the shared memory region.
     *
     * Note we must increment the value of ptr after each write.
     */
    sprintf(ptr,"%s",message0);
    ptr += strlen(message0);
    sprintf(ptr,"%s",message1);
    ptr += strlen(message1);
    sprintf(ptr,"%s",message2);
    ptr += strlen(message2);

    return 0;
}
```


POSIX 공유 메모리 예 - shcons.c

```
// shcons.c

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>

int main()
{
    const char *name = "OS";
    const int SIZE = 4096;

    int shm_fd;
    void *ptr;
    int i;

    /* open the shared memory segment */
    shm_fd = shm_open(name, O_RDONLY, 0666);
    if (shm_fd == -1) {
            printf("shared memory failed\n");
            exit(-1);
    }
```

```
    /* now map the shared memory segment in the address space of the process */
    ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
            printf("Map failed\n");
            exit(-1);
    }

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

    /* remove the shared memory segment */
    if (shm_unlink(name) == -1) {
            printf("Error removing %s\n",name);
            exit(-1);
    }

    return 0;
}
```


POSIX 공유 메모리 실행 예

- 컴파일 시 rt (run time) 라이브러리 지정

```
lee@leeVB:~/os$ gcc -o shprod shprod.c -lrt
lee@leeVB:~/os$ gcc -o shcons shcons.c -lrt
shcons.c: In function ‘main’:
shcons.c:32:2: warning: format ‘%s’ expects argument of type ‘char *’, but arg
ument 2 has type ‘void *’ [-Wformat]
lee@leeVB:~/os$
lee@leeVB:~/os$ ./shprod
lee@leeVB:~/os$ ./shcons
Hello Soonchunhyang Computer Engineering and Science!
lee@leeVB:~/os$
```

실습 과제 - 공유 메모리

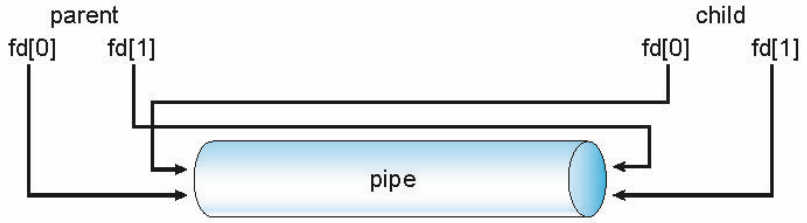
- 앞에서 소개된 POSIX 공유 메모리 프로그램 (shprod.c, shcons.c,그림 3.16, 3.17)을 실행하고 결과를 분석하여라.

파이프 (Pipe)

- 파이프는 두 개의 프로세스가 서로 통신 가능하도록 전달자 역할을 함
- 파이프 구현 시 고려 사항
  - 파이프가 단방향 또는 양방향 통신인가?
  - 양방향인 경우 반이중(half duplex) 또는 전이중(full duplex)인가?
  - 통신하는 두 프로세스 간에 부모-자식과 같은 특정한 관계가 존재하는가?
  - 파이프가 네트워크을 통하여 통신 가능한가?
- 일반파이프(ordinary pipe)
  - 생성한 프로세스 이외에는 접근할 수 없음
  - 일반적으로 부모 프로세스가 파이프를 생성하고, 파이프를 사용하여 자식 프로세스와 통신
- 지명 파이프(named pipe)
  - 부모-자식 관계 없이 프로세스들 간에 접근 가능

일반 파이프 (Ordinary Pipe)

- 생산자-소비자 형태로 두 프로세스 간의 통신을 허용
  - 생산자는 파이프의 한 종단(쓰기 종단, write-end)
  - 소비자는 다른 종단(읽기 종단, read-end)에서 읽음
- 일반 파이프는 단방향 통신만 가능
- 통신 프로세스들 간에 부모-자식 관계
  - fd[0]: 읽기 종단, fd[1]: 쓰기 종단



- 윈도우 시스템에서는 익명 파이프(anonymous pipe)라 함


```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#define BUFFER_SIZE 25
#define READ_END    0
#define WRITE_END   1

int main(void)
{
      char write_msg[BUFFER_SIZE] = "Greetings";
      char read_msg[BUFFER_SIZE];
      pid_t pid;
      int fd[2];

      /** create the pipe */
      if (pipe(fd) == -1) {
            fprintf(stderr,"Pipe failed");
            return 1;
      }

      /** now fork a child process */
      pid = fork();
      if (pid < 0) {
            fprintf(stderr, "Fork failed");
            return 1;
      }
```

```
    if (pid > 0) {  /* parent process */
            /* close the unused end of the pipe */
            close(fd[READ_END]);

            /* write to the pipe */
            write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

            /* close the write end of the pipe */
            close(fd[WRITE_END]);
    }
    else { /* pid == 0, child process */
            /* close the unused end of the pipe */
            close(fd[WRITE_END]);

            /* read from the pipe */
            read(fd[READ_END], read_msg, BUFFER_SIZE);
            printf("child read %s\n",read_msg);

            /* close the write end of the pipe */
            close(fd[READ_END]);
    }

    return 0;
}
```


지명 파이프 (Named Pipe)

- 지명 파이프는 보통의 파이프보다 좀 더 강력한 기능 제공
- 양방향 통신 가능
- 부모-자식 관계 없이 프로세스들 간에 접근 가능
- 여러 프로세스들이 지명 파이프를 사용하여 통신 가능
- UNIX, 윈도우 모두 지명 파이프 지원
  - UNIX에서는 FIFO라고 함

실습 과제 - 일반 파이프

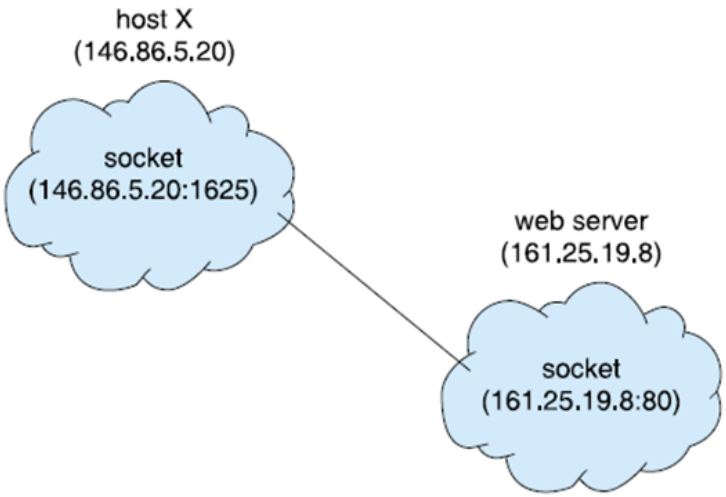
- 앞에서 소개된 일반 파이프 프로그램(그림 3.21, 3.22)을 실행하고 결과를 분석하여라.

클라이언트-서버 통신 (Client-Server Communication)

- 소켓 (socket)
- 원격 프로시저 호출 (Remote Procedure Calls, RPC)

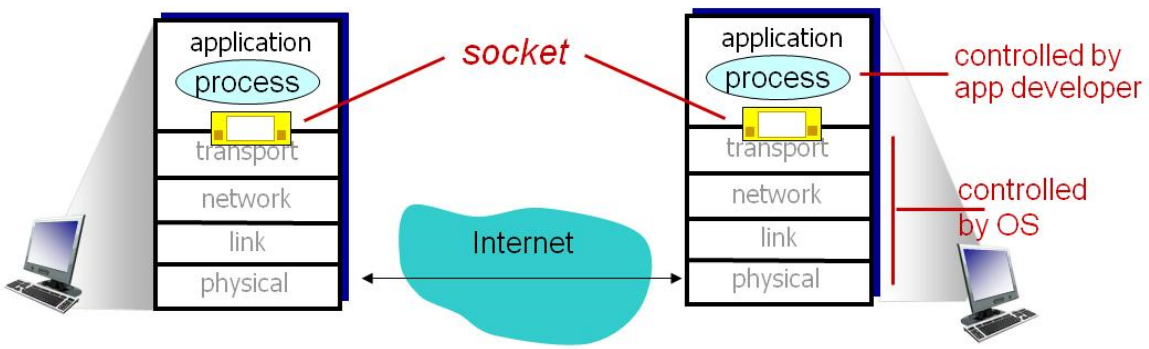
소켓 (Socket) (1)

- 소켓(socket)은 통신의 극점(endpoint)을 정의
  - 소켓은 IP 주소와 포트 번호 두 가지를 접합(concatenate)해서 구별
  - 두 프로세스의 네트워크 통신에 각각 하나씩 두 개의 소켓이 필요



소켓 (Socket) (2)

- 프로세스는 소켓(socket)을 통해 네트워크로 메시지를 송수신
  - 소켓은 호스트의 애플리케이션 계층과 트랜스포트 계층 간의 인터페이스
  - 프로세스는 집(house), 소켓은 출입구(door)에 비유
    - 송신 프로세스는 출입구(소켓) 바깥 네트워크로 메시지를 밀어냄
  - 소켓은 애플리케이션과 네트워크 사이의 API(Application Programming Interface)

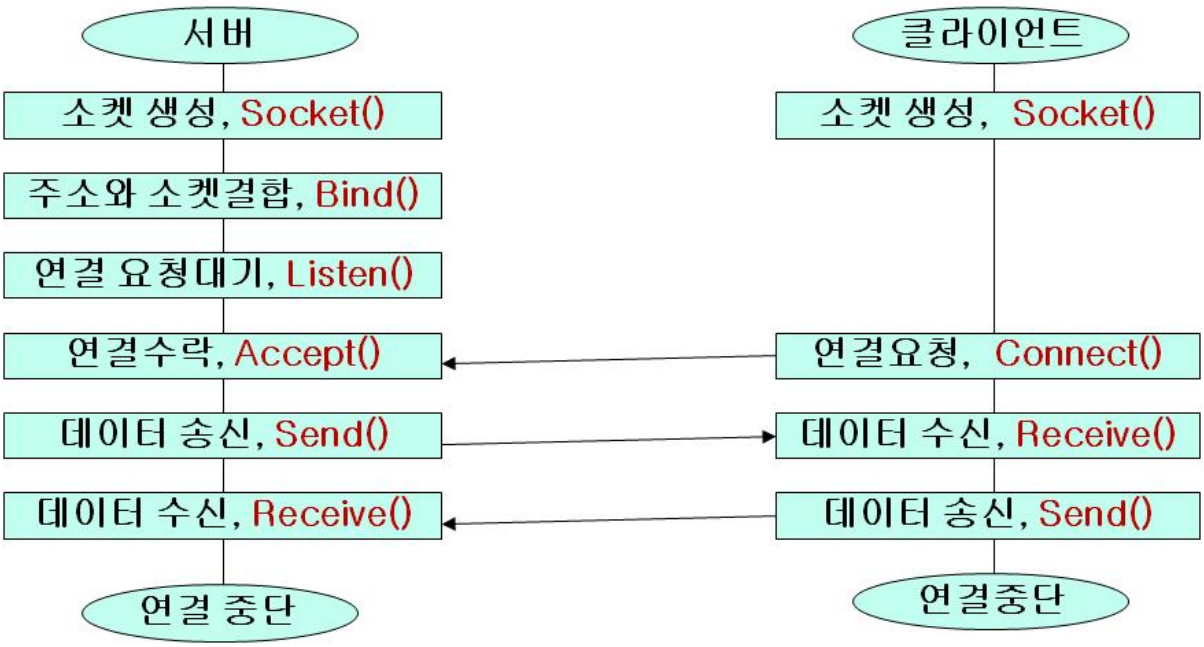


소켓 프로그래밍

- 두 가지 형태의 트랜스포트 서비스
  - UDP: 비연결형, 비신뢰적인 데이터 전송
  - TCP: 연결형, 신뢰적인 데이터 전송

- TCP 소켓 프로그래밍
  - *서버* 프로세스가 먼저 수행 중에 있어야 함
    - 서버는 클라이언트의 초기 접속을 처리하는 소켓을 생성해야 함
  - *클라이언트*는 서버에 초기 접속
    - 클라이언트는 TCP 소켓을 생성하고, 서버 프로세스의 IP 주소와 포트 번호를 명시하여 서버에 접속
  - *서버*는 클라이언트에 의해 초기 접속 클라이언트와 통신하는 서버 프로세스를 위한 새로운 소켓(연결 소켓)을 생성
  - *서버*와 *클라이언트*가 데이터 송수신

TCP 클라이언트/서버 소켓 상호동작



Java 소켓 예

- Java는 세 가지 종류의 소켓을 제공
  - 연결 기반(TCP) 소켓은 Socket 클래스로 구현
  - 비연결성(UDP) 소켓은 DatagramSocket 클래스를 사용
  - 멀티캐스트 소켓은 MulticastSocket 클래스 사용
    - DatagramSocket 클래스의 서브클래스
- 예제 프로그램
  - 연결 기반 TCP 소켓을 사용하는 date 서버
  - 클라이언트는 서버로부터 현재 날짜와 시간을 요청
  - 서버는 클라이언트에게 현재 날짜와 시간을 보내줌
    - 서버는 포트 6013을 listen

Date 서버

```
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();

            PrinterWriter 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);
      }
   }
}
```

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

public class Date Client
{
    public static void main(String[] args) {
        try {
            //make connection to server socket
            Socket sock = new Socket("127.0.0.1", 6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            // read the date from the socket
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            //close the socket connection
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Date 클라이언트

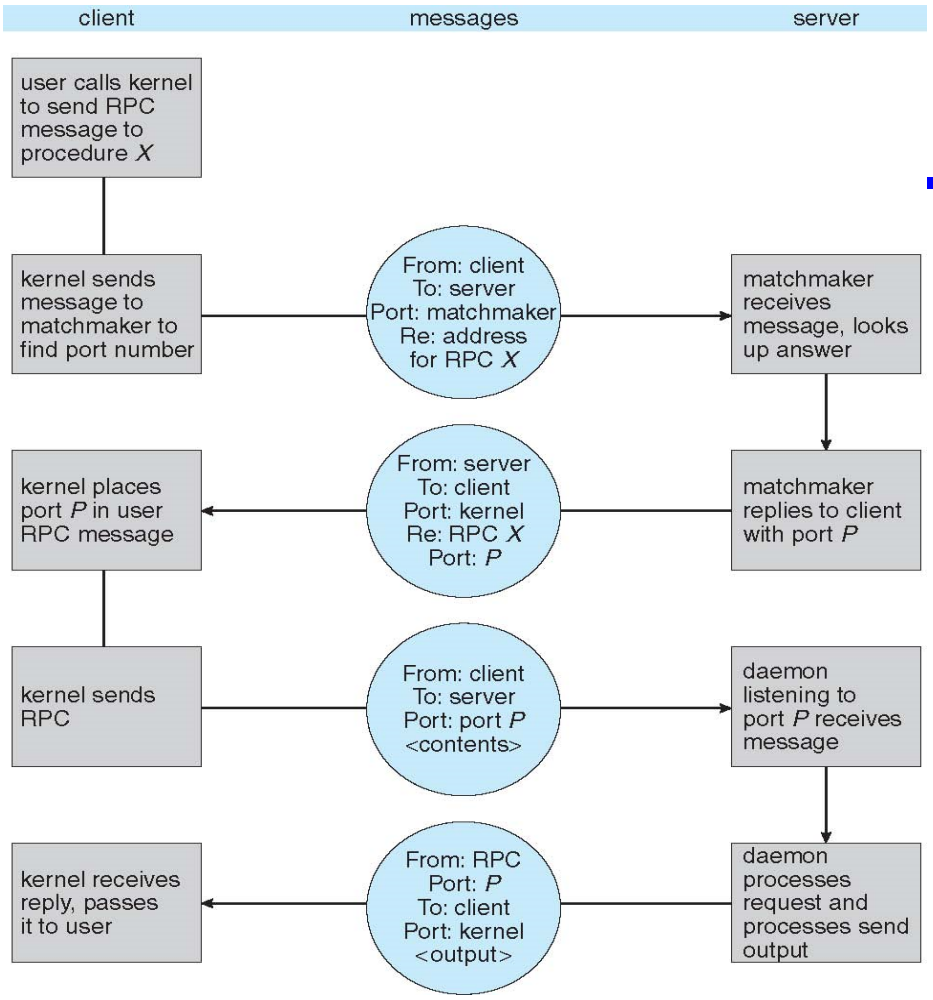
소켓 장단점

- 소켓을 이용한 통신은 분산된 프로세스들 간에 널리 사용되고 효율적
- 소켓 통신은 저수준 통신
  - 소켓은 쓰레드들 간에 구조화되지 않은 바이트 스트림만을 통신
  - 원시적인 바이트 스트림 데이터를 구조화하여 해석하는 것은 클라이언트와 서버의 책임
- 소켓 통신의 대안으로 보다 고수준 통신
  - 원격 프로시저 호출(Remote Procedure Call, RPC)
  - 원격 메서드 호출(Remote Method Invocation, RMI)

원격 프로시저 호출 (Remote Procedure Call, RPC)

- RPC는 네트워크에 연결되어 있는 두 시스템 사이의 통신을 위해 프로시저 호출을 추상화하기 위한 방편으로 설계
  - 클라이언트가 원격 호스트의 프로시저 호출하는 것을 마치 자기의 프로시저처럼 호출
- 원격 프로시저마다 다른 스텁(stub)이 존재
  - 스텁은 원격 호출을 대행해 주는 프록시(proxy)
  - 클라이언트 측 스텁
    - 원격 서버의 포트를 찾고 매개변수를 정돈(marshall)
      - 매개변수 정돈(parameter marshalling)이란 프로시저에게 갈 매개변수를 네트워크로 전송하기 위해 적절한 형태로 재구성하는 작업
    - 스텁은 메시지 전달 기법을 사용하여 서버에게 메시지를 전송
  - 서버 측 스텁
    - 메시지를 수신한 후 정돈된 매개변수를 해제
    - 서버의 프로시저를 호출
- 운영체제는 미리 고정된 RPC 포트를 통해 랑데뷰용 디먼(matchmaker라 불림)을 제공

RPC 수행



과 제

fork 실습과제

- p.132 그림 3.8의 프로그램을 일부 수정한 프로그램(fork.c)이다. 다음과 같이 두 개의 터미널에서 실행하고 결과를 분석하여라.
  - 터미널1에서 소스 작성하고 실행 시작 후 문자 입력 전 대기
  - 터미널2를 생성하고 다음을 실행하고 출력 분석
    $ ps –al
  - 터미널 1에서 문자 입력하고 실행 결과 분석

- VirtualBox 과제 캡처 시 주의 사항
  - 사용자 셸 프롬트에 자기 이름 표시
  - 프롬트 변경 예
    $ export PS1="(이름)$ "

```c
#include <stdio.h>

int main( )
{
  int pid;   /* pid_t pid; */

  /* 새 프로세스를 생성한다(fork) */
  pid = fork();

  /* 입력 대기 */
  printf("Input any character to continue\n");
  getchar();

  if (pid < 0) { /* 오류가 발생했음 */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if (pid == 0) { /* 자식 프로세스 */
    execlp("/bin/ls", "ls", NULL);
  }
  else { /* 부모 프로세스 */
    /* 부모가 자식이 완료되기를 기다릴 것임 */
    wait(NULL);
    printf("Child Complete");
    exit(0);
  }
}
```



```
lee@leeVB: ~/os_test
lee@leeVB:~/os_test$ cc -o fork fork.c
fork.c: In function ‘main’:
fork.c:16:7: warning: incompatible implicit declaration of built-in function ‘exit’ [ena
bled by default]
fork.c:19:7: warning: incompatible implicit declaration of built-in function ‘execlp’ [e
nabled by default]
fork.c:25:7: warning: incompatible implicit declaration of built-in function ‘exit’ [ena
bled by default]
lee@leeVB:~/os_test$
lee@leeVB:~/os_test$ ./fork
Input any character to continue
Input any character to continue
```

```
lee@leeVB: ~/os_test
lee@leeVB:~/os_test$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2847  1885  0  80   0 -   503 n_tty_ pts/1    00:00:00 fork
1 S  1000  2848  2847  0  80   0 -   503 n_tty_ pts/1    00:00:00 fork
0 R  1000  2849  2589  0  80   0 -  1482 -      pts/3    00:00:00 ps
lee@leeVB:~/os_test$
```

```
lee@leeVB: ~/os_test
lee@leeVB:~/os_test$ cc -o fork fork.c
fork.c: In function ‘main’:
fork.c:16:7: warning: incompatible implicit declaration of built-in function ‘exit’ [ena
bled by default]
fork.c:19:7: warning: incompatible implicit declaration of built-in function ‘execlp’ [e
nabled by default]
fork.c:25:7: warning: incompatible implicit declaration of built-in function ‘exit’ [ena
bled by default]
lee@leeVB:~/os_test$
lee@leeVB:~/os_test$ ./fork
Input any character to continue
Input any character to continue

h
fork    fork.c~    shm-posix.c   shm-posix2    shm-posix2.c~
fork.c  shm-posix  shm-posix.c~  shm-posix2.c  shm1.c
Child Completelee@leeVB:~/os_test$
```

```
lee@leeVB: ~/os_test
lee@leeVB:~/os_test$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2847  1885  0  80   0 -   503 n_tty_ pts/1    00:00:00 fork
1 S  1000  2848  2847  0  80   0 -   503 n_tty_ pts/1    00:00:00 fork
0 R  1000  2849  2589  0  80   0 -  1482 -      pts/3    00:00:00 ps
lee@leeVB:~/os_test$
lee@leeVB:~/os_test$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 R  1000  2850  2589  0  80   0 -  1474 -      pts/3    00:00:00 ps
lee@leeVB:~/os_test$
```


fork 연습문제

❑ 연습문제 3.1

- 그림 3.30 프로그램을 앞의 실습과 같이 프로그램을 수정하여 실행 후 실행 결과 분석

실습 과제 - 공유 메모리, 일반 파이프

- 앞에서 소개된 POSIX 공유 메모리 프로그램 (shprod.c, shcons.c,그림 3.16, 3.17)을 실행하고 결과를 분석하여라.
- 앞에서 소개된 일반 파이프 프로그램(그림 3.21, 3.22)을 실행하고 결과를 분석하여라.

특별 실습 과제

- POSIX 공유 메모리(또는 파이프)를 사용하여 유한 버퍼의 생산자, 소비자 프로그램을 작성하여라
  - 생산 및 소비되는 데이터는 임의의 응용 데이터
  - 소스 프로그램 및 설명
  - 실행 예