운영체제의 입장에서, 프로세스란

휴먼지능정보공학과

201910798

노희재

1.프로세스의 개념

2.프로세스의 상태변화

3.프로세스의 관리

4.프로세스 스케쥴링

5.프로세스 간 통신

서론

1.프로세스의 개념 Process Concept

운영체제의 입장에서, 프로세스란 무엇인가?

Essay를 쓰기 전 운영체제에 대한 저의 생각을 간단히 서술하겠습니다. 운영체제를 공부하기 위해 많은 교재들을 읽었습니다. 강의 자료, 수업서, 인터넷 강의 등을 보면서 운영체제가 왜 어려운지에 대해 알 것 같습니다. 책들의 과목을 분류한 대분류 항이나 강좌의 달라도 내용의 순서는 거의 유사해 있습니다. 하지만 운영체제의 프로세스 부분을 읽을 때마다 설명하는 내용의 순서가 달랐습니다. 이는 많은 내용이 복잡하고 서로밀접 연관되어 있기 때문입니다. 그래서 저자의 흐름보다는 책들의 한가지씩 관점에 이해하고 넘어가서 흐름을 그려본 후에 공부했고 이러한 방향으로 Essay를 작성하겠습니다.

먼저 프로세스와 같은 의미를 가지는 용어들이 있습니다. 거의 같은 의미라 볼 수 있으나 시대가 지나면서 내용이 확장되거나 변화에 따라 용어들이 생겨났다고 볼 수 있습니다.

작업(job) → 사용자 프로그램(user program) or task = process

초창기 컴퓨터는 일괄처리 시스템(batch system)으로 작업을 실행했습니다. 그 뒤를 이어 시분할시스템(time-shared system)으로 사용자 프로그램이나 타스크를 실행했습니다.

위의 내용처럼 컴퓨터에서 실행하는 활동들을 프로세스라 볼 수 있습니다. 현대에는 프로세서라는 용어를 선호하지만 작업이라는 용어는 운영체제에서 역사적으로 중요한 의미가 있으므로 두 용어는 거의 interchangeably 합니다.

프로그램은 실행될 때 작업을 수행하는 명령어들의 집합으로 볼 수 있습니다. 즉 프로세스란 쉽게 말해 실행중인 프로그램입니다.

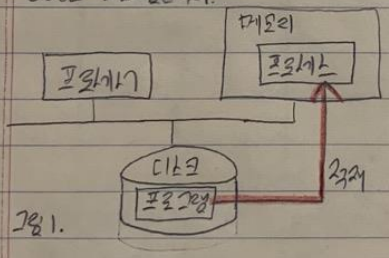


그림 1.

그림 1. 에서 볼 수 있듯이 디스크에 실행된 프로그램을 실행하면, 메모리에 적재되면서 프로세스가 됩니다. 프로그램의 실행은 마우스 클릭과 같은 GUI 나 명령창에 프로그램을 입력하는 등의 방법이 있습니다. 중요한 것은 프로그램 그 자체는 프로세스가 아니라는 것을 기억해야 합니다. 프로그램은 명령어 리스트를 내용으로 가진 디스크에 저장된 파일과 같은 수동적인 존재(passive entity)인 반면 프로세스는 능동적인 존재(active entity) 입니다.

프로세스의 종류.

구분	종류	설명
목적	시스템(커널) 프로세스	모든 시스템의 메모리와 프로세서의 명령에 액세스할 수 있는 프로세스이다. 프로세스 실행 순서를 제어하거나 다른 사용자 및 커널 영역을 침범하지 못하게 감시한다.
	사용자 프로세스	사용자 코드를 실행하는 프로세스이다. 사용자 프로세서로 실행할 수 있다.
병행 수행 방법	독립 프로세스	다른 프로세스에 영향을 주지 않거나 다른 프로세스의 영향을 받지 않으면서 실행한다.
	협력 프로세스	다른 프로세스에 영향을 주거나 다른 프로세스에게 영향을 받는 병행 프로세스이다.

앞서, 프로세스는 프로그램이 메모리에 적재되면서 만들어진다고 했습니다. 이제 프로그램이 프로세스가 될 때, 갖게 되는 메모리 구조에 대해 알아보겠습니다.

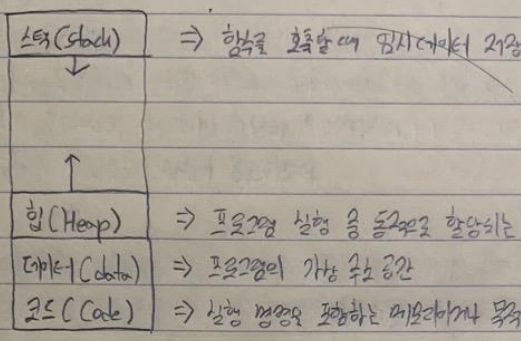


그림 2. 일반적인 메모리 구조

그림 2. 는 프로세스의 일반적인 메모리 구조를 나타냅니다. 데이터와 코드 섹션의 크기는 고정되기 때문에 프로그램 실행 시간 동안 크기가 변하지 않지만 스택 및 힙 섹션은 프로그램 실행 중에 동적으로 늘어나거나 줄어들 수 있습니다. 스택 및 힙 섹션이 서로 반대 방향으로 커지더라도 운영체제는 서로 겹치지 않도록 해야 합니다. 아래의 그림 3. 은 메모리 공간 내 C 프로그램의 배치를 보여주고 있습니다. 그림 2. 의 일반적인 메모리 구조와 비슷하지만 몇 가지 차이점이 있습니다.

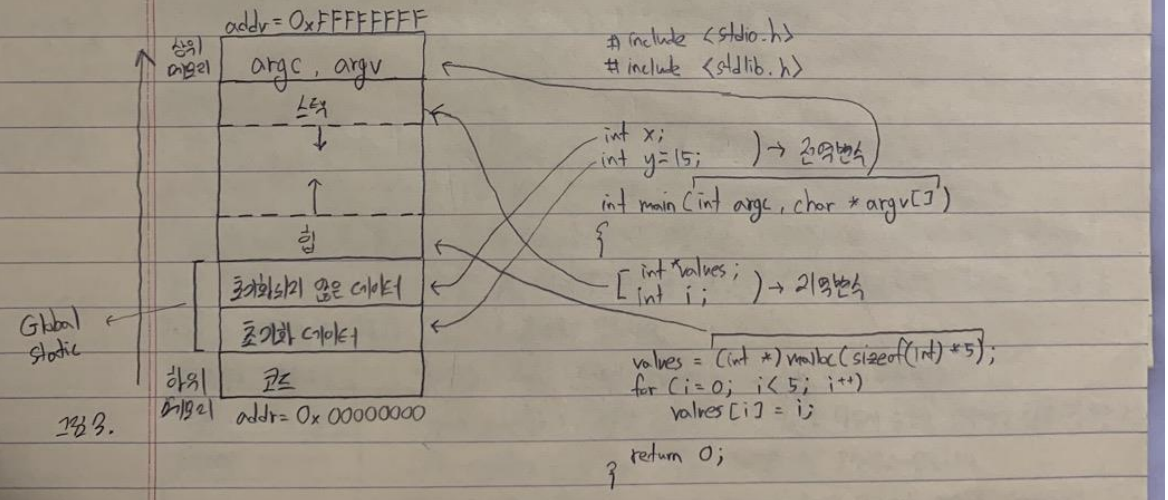


그림 3.

그림 2. 와 그림 3. 을 통해 차이점을 알 수 있습니다. 전역 데이터 섹션 (Global static)은 초기화된 데이터와 초기화되지 않은 데이터 섹션으로 나누어집니다. 또한 main() 함수에 전달된 argc, argv 매개변수가 저장되는 별도의 섹션이 있습니다. 이어서 2. 프로세스의 상태 변화 에 대해 알아보겠습니다.

2.프로세스의 상태변화 Process State

2.1.프로세스의 상태

프로세스는 실행되면서 그 상태(state)가 변합니다. 프로세스는 다음 상태 중 한 가지에 있게 됩니다.

- new : 프로세스가 생성 중이다.
- running : 명령어들이 실행되고 있다.
- waiting : 프로세스가 어떤 이벤트(입출력 완료 또는 신호의 수신 같은)가 일어나기를 기다린다.
- ready : 프로세스이 처리기에 할당되기를 기다린다.
- terminated : 프로세스의 실행이 종료되었다.

이 상태의 이름은 임의적이고 운영체제마다 변화되며 이 상태들은 모든 시스템에서 찾아볼 수 있습니다.
중요한 것은 어느 한순간에 하나의 처리기 코어에서는 오직 하나의 프로세스만이 실행된다는 것입니다.
그림 4는 프로세스의 상태 변화 다이어그램 입니다.

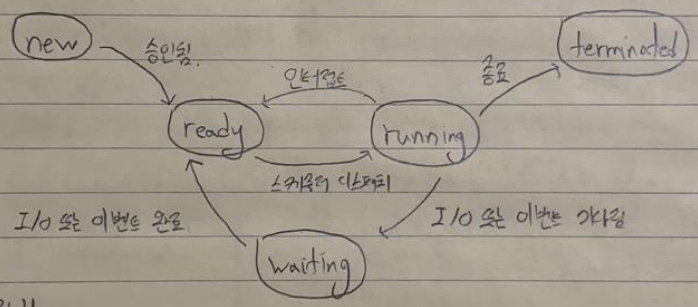


그림 4.

2.2. PCB

프로세스가 생성되면 메모리에 프로세스 제어 블록(PCB, Process Control Block)이 생성됩니다.
이 프로세스 제어블록은 특정 프로세스 정보를 저장하는 데이터 블록 또는 레코드로 볼 수 있으며
작업 제어 블록(TCB, Task Control Block)이라고도 불립니다. 운영체제가 프로세스 제어에 필요한
상태 정보를 저장하며, 프로세스가 실행 종료되면 해당 프로세스 제어 블록도 삭제됩니다. 아래의 그림 5.는
PCB를 나타내며 다음과 같은 것들을 포함하고 있습니다.

process state
process number
process counter
registers
memory limits
list of open files
. . .

그림 5.

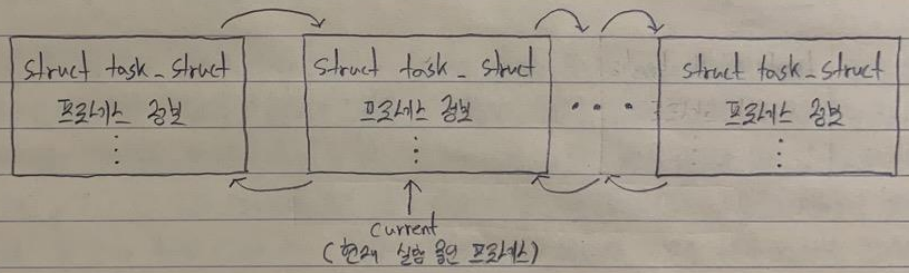
- Process state : 상태는 running, waiting 등이 있습니다.
- Program counter : 프로그램 카운터는 이 프로세스가 다음에 실행할 명령어의 주소를 가리킵니다.
- CPU registers : process-centric 레지스터들을 포함하고 있습니다.
- CPU scheduling information : 프로세스 우선순위, 스케줄 큐에 대한 포인터를 포함합니다.
- Memory-management information : 프로세스에 할당된 메모리 정보를 포함합니다.
- Accounting information : CPU 사용시간과 경과된 실시간, 시간 제한 정보를 포함합니다.
- I/O status information : 프로세스에 할당된 입출력 장치들과 열린 파일의 목록 등을 포함합니다.

운영체제는 프로세스 간의 불록은 약간의 회계 데이터와 함께 프로세스를 시작시키거나 다시 시작하게에 필요한 모든 데이터를 위한 저장소의 역할을 합니다. 이에서 실제 프로그램에서 프로세스 간의 불록의 구현을 살펴보겠습니다. 아래는 Linux 운영체제의 PCB 구현으로 C 구조체 task_structure로 표현됩니다.

```
pid_t pid;                      /* 프로세스 식별자 */
long state;                     /* 프로세스 상태 */
unsigned int time_slice;        /* 스케줄링 정보 */
struct task_struct *parent;     /* 이 프로세스의 부모 */
struct list_head children;      /* 이 프로세스의 자식들 */
struct files_struct *files;     /* 오픈 파일 */
struct mm_struct *mm;           /* 이 프로세스의 주소공간 */
```

이 프로세스의 상태는 이 구조체 필드 long state 에 의해 표시됩니다. Linux 커널 안에서 모든 활성 프로세스들은 task_structure의 이중 연결 리스트로 표현됩니다. 다음 그림6. 에서 보이는 것처럼 커널은 현재 실행 중인 프로세스를 가리키는 포인터를 유지합니다. (current)

그림6.



현재 실행 중인 프로세스의 상태를 new_state로 바꾸고 한다면 다음과 같은 명령문에 의해 상태가 바뀌게 된다. current → state = new_state;

2.3 Threads

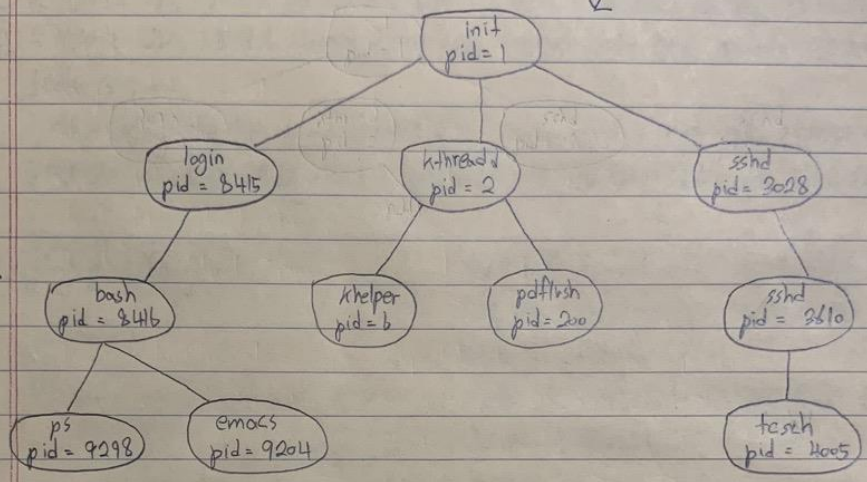
지금까지는 프로세스가 단일 실행 스레드를 가졌습니다. 예를 들자면, 워드 프로세서 프로그램을 실행 시에 단일 스레드는 한 번에 한 가지 일만 처리할 수 있으므로 문자를 입력하면서 동시에 철자 검사기를 실행할 수 없습니다. 다행히 현대의 운영체제는 한 프로세스가 다수의 스레드를 가질 수 있도록 허용합니다.

이러한 다중 실행 스레드 덕분에 워드 프로세서 실행 시에 하나의 스레드에는 사용자의 입력 관리를 맡기고 다른 스레드에는 철자 검사기 실행을 맡겨 수행할 수 있도록 합니다. 다중 스레드를 지원하는 시스템에서는 PCB는 각 스레드에 관한 정보를 포함하도록 확장됩니다.

3.프로세스의 관리

먼저, 프로세스의 구조에 대해서 알아보겠습니다. 프로세스는 실행 중 프로세스 상태 시스템 호출을 이용하여 새로운 생성합니다. 프로세스는 부모-자식 관계를 유지하면서 계층적으로 생성되는데, 이 때 생성하는 프로세스를 부모 프로세스, 생성되는 프로세스를 자식 프로세스라고 합니다. 부모 프로세스는 자식 프로세스의 생성 과정을 반복하면서 계층 구조를 형성합니다. 아래의 그림 7. 은 Linux의 프로세스 트리를 나타냅니다.

그림 7.



이 프로세스 트리를 간단히 살펴보겠습니다. UNIX, Linux 및 Windows와 같은 대부분의 현대 운영체제는 유일한 프로세스 식별자(pid)를 사용하여 구별하고 이 식별자는 보통 정수입니다. pid는 시스템의 각 프로세스에 고유한 값을 가리키도록 할당되고 커널이 유지하고 있는 프로세스의 다양한 속성에 접근하기 위한 Index로 사용됩니다.

항상 pid=1 인 init 프로세스가 모든 사용자 프로세스의 루트 부모 프로세스 역할로 생성하고 시스템이 부팅될 때 생성되는 첫 번째 사용자 프로세스이다. 시스템이 부팅되면 init 프로세스는 다양한 사용자 프로세스로 생성하는데 그림 7. 에서는 login, kthreadd, sshd 프로세스를 생성한 것을 볼 수 있다.

3.1 프로세스 생성

먼저 프로세스의 생성시기를 본다면 시작 프로그램 같은 일괄처리 환경일 경우 작업이 요청될 때 프로세스를 생성하고 대화형 환경이라면 새로운 사용자가 로그온(log-on)할 때 프로세스를 생성합니다.

프로세스 생성시 필요한 세부 작업 과정은 아래와 같습니다.

1. 새로운 프로세스에 프로세스 식별자(process ID) 할당
2. 프로세스의 모든 구성요소를 포함할 수 있는 주소 공간과 프로세스 제어 블록 공간 할당
3. 프로세스 제어 블록 초기화 (프로세스 상태, 프로그램 카운터 등 초기화, 사용자 요청, 프로세스 제어 정보(권한 등)를 포함)
4. 링크 (적당 큐에 삽입)

프로세스가 새로운 프로세스를 생성할 때, 두 프로세스를 실행시키는 데 두 가지 가능한 방법이 존재합니다.

① 첫 번째는 부모 프로세스와 자식 프로세스를 동시에 실행, 즉 병행하게 실행하는 것 입니다.

② 두 번째는 부모 프로세스는 일부 또는 모든 자식 프로세스가 실행을 종료할 때까지 대기하는 것입니다.

새로운 프로세스들의 Address Space(주소공간) 측면에서 본다면 다음과 같은 두 가지 가능성이 있습니다.

① 첫 번째로 자식 프로세스는 부모 프로세스의 복사본으로 자식은 부모와 똑같은 프로그램 데이터를 가지는 것입니다.

② 두 번째는 자식 프로세스가 자신에게 적재될 새로운 프로그램을 가지는 것으로 나름대로의 프로그램을 소유하는 것입니다.

이들의 차이점을 보기 위해 UNIX의 시스템 콜과 Windows API를 이용한 새로운 프로세스의 생성에 대해 살펴보겠습니다. 먼저 UNIX 시스템 콜 방식을 보면 C 프로그램을 실행할 것이며 코드는 아래와 같습니다.

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

int main()
{
pid_t pid;
    /* 새 프로세스를 생성한다(fork) */
    pid = fork();

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

새로운 프로세스는 fork() 시스템 콜로 생성되며, exec() 시스템 콜은 fork() 이후에 자신의 메모리 공간을 새로운 프로그램으로 고치입니다.

윗쪽의 코드를 보면 동일한 프로그램의 복사본을 실행하는 두 개의 서로 다른 프로세스를 갖습니다. 이때 자식 프로세스는 exec()의 한 버전인 execlp() 시스템 콜을 사용하여 자신의 값 공간을 UNIX 명령 /bin/ls 로 덮어씁니다. 부모는 wait() 시스템 콜로 자식 프로세스의 끝나기를 기다립니다. 자식 프로세스가 끝나면, 부모 프로세스는 wait() 호출로부터 깨어나며, exit() 시스템 콜을 사용하여 끝낸다. 이러한 과정은 그림 8. 에 묘사되었습니다.

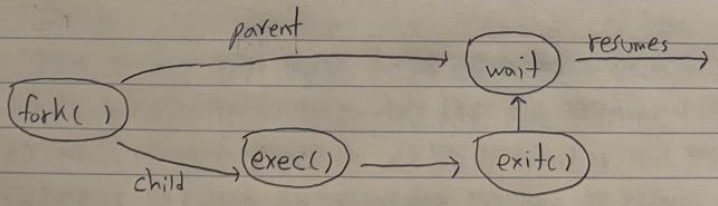


그림 8.

위의 시나리오에서는 부모와 자식이 같은 코드를 실행하는 병행 실행 프로세스입니다.
또한 자식은 부모의 복사본이기 때문에 각 프로세스는 모든 데이터에 대해 자신만의 복사본을 가지고 있습니다.

다른 예로 Windows 에서의 프로세스 생성을 살펴볼 것이며 코드는 아래와 같습니다.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
 STARTUPINFO si;
 PROCESS_INFORMATION pi;
    /* 메모리 할당 */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));
    /* 자식 프로세스 생성 */
    if(!CreateProcess(NULL, /* 명령어 라인 사용 */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* 명령어 라인 */
     NULL, /* 프로세스를 상속하지 말것 */
     NULL, /* 스레드 핸들을 상속하지 말 것*/
     FALSE, /* 핸들 상속 디스에블 */
     0,   /* 생성 플래그 없음 */
     NULL, /* 부모 환경 블록 사용 */
     NULL, /* 부모 프로세스가 존재하는 디렉터리 사용 */
     &si
     &pi))
    {
     fprintf(stderr, "Create Process failed");
     return -1;
    }
    /* 부모 프로세스가 자식 프로세스의 끝나길 기다림*/
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");
    /* 핸들 닫기*/
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

위의 코드는 mspaint.exe 프로그램을 적재하는 자식 프로세스의 생성을 하는 C프로그램 코드입니다. fork() 와 다른 CreateProcess() 를 사용하여 프로세스를 생성하고 wait() 대신에 WaitForSingleObject()를 사용해 자식 프로세스가 끝나기를 기다립니다.

3.2프로세스의 종료

프로세스가 마지막 명령을 실행하고, 종료하여 운영체제에게 프로세스의 삭제를 요청합니다. 일괄 처리 환경에서는 주로 종료 코드의 신호를 인터럽트 발생 또는 시스템 호출로 종단을 명령합니다. 대화형 환경에서는 사용자가 그그 오프하거나 터미널을 닫으면 종료합니다. 종료 방법에는 abort 나 exit 명령어로 프로세스를 종료하거나 wait 명령어로 부모 프로세스가 자식 프로세스의 종료를 기다립니다. 부모 프로세스가 자식프로세스를 종료시키는 경우도 있습니다. 보통 부모가 종료하면 운영체제가 자식도 종료하고 이를 연속 종료라 합니다. 또한 자식이 할당된 자원을 초과하여 사용하거나 자식에 할당한 작업이 없을 경우에도 부모의 자식 프로세스가 종료됩니다.

종료 이유에는 프로세스가 운영체제의 서비스 호출로 부터 정상 종료를 하고 이 외에는 시간초과, 산술 오류, 보호 오류, 메모리 오류, 그리고 파일 없거 실패 등이 있습니다.

프로세스가 종료되면 프로세스로 제거됩니다. 이를 프로세스 파괴라고도 합니다. 사용했던 자원을 시스템에 돌려주고, 해당 프로세스는 시스템 리스트나 테이블에서 사라져 프로세스 제어블록을 회수합니다. 프로그램은 여전히 디스크에 저장되며 멀티 운영체제는 부모가 종료되었을 때 자식이 몇개나 있든 상관 없이 부모부터 비롯된 모든 자식 프로세스들은 종료되어야 합니다. 이를 Cascading Termination (연쇄적 종료) 라고 부르며 이 작업은 운영체제에 의해 실행됩니다. 부모 프로세스는 wait() 시스템 콜을 실행하여 자식 프로세스가 종료될 때까지 기다릴 수 있는데 이 때 부모가 자식의 종료 상태를 알아낼 수 있도록 하나의 인자를 전달합니다. 이 시스템 콜은 부모가 어느 자식이 종료되었는지 구별할 수 있도록 종료된 자식의 프로세스 식별자를 반환합니다.

```
pid_t pid;
int status;
pid = wait(&status);
```

종료되었지만 부모 프로세스가 아직 wait() 호출을 하지 않은 프로세스를 Zombie (좀비) 프로세스라고 합니다. 부모 프로세스가 wait() 를 호출하는 대신 종료한다면 그 때의 프로세스를 Orphan (고아) 프로세스라 하고 이 경우 Orphan의 부모를 Init (프로세스 계통구조의 root) 프로세스로 지정하여 Orphan 문제를 해결할 수 있습니다.

Multiprocess architecture Chrome Browser

많은 웹 브라우저들은 단일 프로세스 구조로 운영되었고 멀티 탭은 여전히 그 방식을 사용합니다. 만약 하나의 웹 사이트에 문제가 생기면 전체 브라우저에 영향을 미쳐 종료해야 하므로 이 방식은 여러움이 있습니다. Google의 Chrome 브라우저는 다중 프로세스 구조를 활용하여 이 문제를 해결하기위해 설계되었고 브라우저, 랜더러, 플러그인의 세 유형의 프로세스를 구별하는 것이 방법입니다.

- Browser (브라우저) 프로세스는 사용자 인터페이스와 함께 디스크와 네트워크 입출력을 관리하는 책임을 집니다.
- Renderer (랜더러) 프로세스는 웹 페이지를 표시하기 위한 프로그램 논리를 포함합니다. 새 탭에서 열린 웹 사이트나 새로운 랜더러 프로세스가 생성되고 여러 랜더러 프로세스가 동시에 활동하게 됩니다. 또한 랜더러 프로세스는 샌드박스에서 실행되는데, 이는 보안 취약점의 영향을 최소화하기 위해 디스크와 네트워크 입출력에 대한 접근이 제한된다는 것을 의미한다.
- Plug-in (플러그인) 프로세스는 사용중인 플러그인 종류마다 생성됩니다.

이러한 방식으로 한 웹사이트의 오류이로 다른 프로세스에는 영향을 받지 않을 수 있습니다. 자원이 더 많이 소비되는 다중 프로세스 구조를 활용한 방식이 더 낫다고 생각합니다.

3.3 프로세스의 중단과 재시작

지금까지 프로세스의 생성과 종료에 대해 알아보았는 프로세스는 여러 많은 이유에 중단과 재시작 또한 빈번하게 발생합니다. 먼저 프로세스의 중단은 시스템의 유휴 (노는) 시간 문제를 해결해 줍니다. 운영체제가 실행 중인 프로세스를 중단했다가 다시 실행하는 방법을 이용하면 시스템 전체의 복하를 증가시키기 위고 프로세스에 서비스를 제공할 수 있습니다. 사례로 보면 다중 프로그래밍에서의 중단은 프로세스 입출력 요구 외에 다른 원인으로

프로세스가 실행을 중단한 상태일 때 발생하고 단일 처리 시스템에서는 해당 프로세스가 스스로 중단하는 경우다. 다중 처리 시스템에서는 다른 프로세서가 실행 중인 프로세스를 중단한다. 중단된 프로세스는 다른 프로세서가 즉시실행의 준비는 실행이 불가하며 강제로 중단 시 해당 프로세스에 할당된 자원을 반환하고, 자원의 상황에 따라 반환 여부를 결정한다. 메인 메모리에서는 프로세서 중단 즉시 반환하고 보조 메모리에서는 중단 시점을 예측할 수 없기나 너무 길 때 반환합니다. 중단된 프로세스는 중단된 시점부터 다시 시작합니다. 아래의 그림 9. 는 중단과 재개를 거치는 프로세스의 상태 변화입니다.

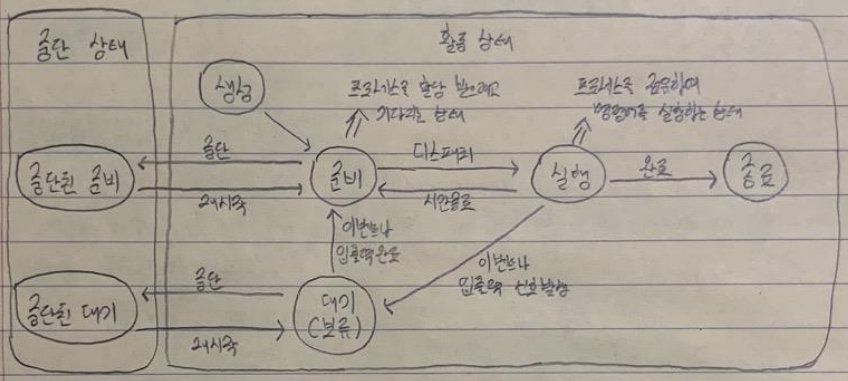


그림 9.

4.프로세스 스케쥴링

다중 프로그래밍의 목적은 CPU 이용을 최대화하기 위함에 항상 어떤 프로세스가 실행되도록 하는데 있습니다. Degree of multiprograming, 다중 프로그램 정도는 현재 메모리에 있는 프로세스의 수를 나타내며 이것을 제어하는 것은 Long-term Scheduler (혹은 Job scheduler) 입니다. 장기 스케줄러는 준비큐에 어떤 프로세스가 들어갈지 결정하므로 수행 빈도가 적고 느립니다.

시분할의 목적은 각 프로그램이 실행되는 동안 사용자와 상호작용할 수 있도록 프로세스를 상태에서 CPU 코어를 빈번하게 교체하는 것입니다. 이 목적을 위해 Short-term scheduler (혹은 CPU scheduler)가 다음에 실행될 프로세스를 선택하고 CPU에 할당합니다. 단기 스케줄러는 적어도 하나의 시스템에 한 개만 있을 수 있고 자주 수행되어 빠릅니다.

일반적으로 대부분의 프로세스는 I/O-bound process 또는 CPU-bound process로 설명할 수 있습니다.

I/O-bound process는 계산 보다 I/O 에 더 많은 시간을 소비하는 프로세스이고 CPU-bound process는 계산에 더 많은 시간을 소비하며 I/O 요청은 자주 상황의 많습니다. 입출력 중심 프로세스는 프로세서를 짧게 자주 사용하도록 하고, CPU 중심 프로세스는 CPU를 길게 사용하되 사용횟수를 줄여 균형을 유지합니다. 이러한 방식을 이용해 장기 스케줄러는 좋은 프로세스 mix를 위해 노력해야 한다. 시스템에는 다른 큐들 존재합니다.

- Job Queue - 시스템에 있는 모든 프로세스의 집합
- Ready Queue - main memory에 있는 모든 프로세스의 집합으로 준비와 대기 상태에 있음.
- Device Queues - I/O 장치를 기다리는 프로세스의 집합. / 프로세스는 다양한 큐를 거치한다.

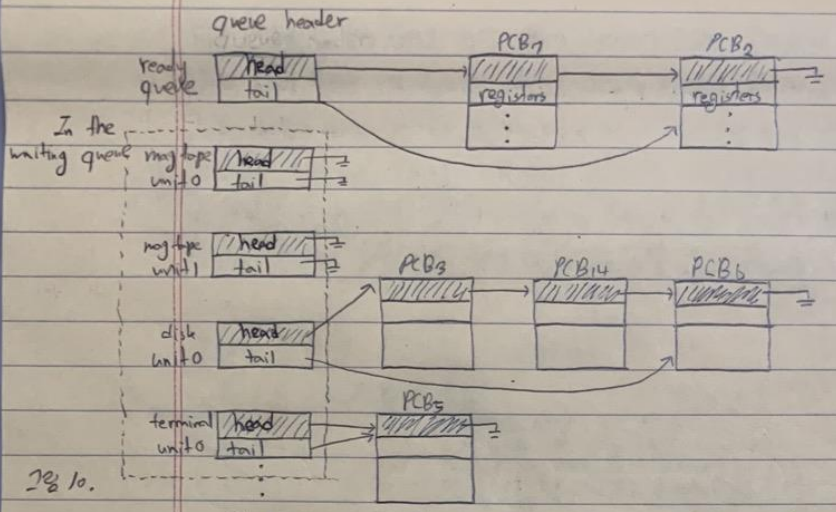


그림 10.

위의 그림 10. 은 플레쉬와 다양한 입출력 장치 대기 큐를 보여줍니다. 아래의 그림 11.은 프로세스 스케줄링을 나타내는 큐잉 다이어그램입니다.

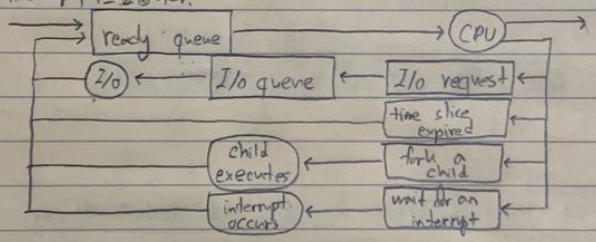


그림 11.

Multitasking in mobile systems

멀티 모바일 시스템은 다른 프로세스는 많은 수로 하나의 프로세서만 돌아갈 수 있도록 합니다. 스크린의 한계 때문에 foreground와 background process를 지원합니다. foreground process 은 사용자 인터페이스를 통해 제어하는 것이고 background process는 활동은 하고 있지만 화면에 보이지 않는 것입니다. background process는 한계를 가지고 있는데 이 한계는 오디오 실행 같은 long 실행 task, 이벤트의 알림을 받는 것 등이 있습니다. Android는 이런 한계를 위해 service를 사용하고 service는 background 프로세스가 멀티태스킹 환경에서 좋은 메모리의 사용과 인터페이스를 가지고 있지 않습니다.

4.1 문맥 교환

Context Switch

CPU가 다른 프로세스로 전환할 때, 문맥교환을 통해 시스템은 이전 프로세스의 상태를 저장하고 새로운 프로세스의 저장된 상태를 적재해야만 합니다. 프로세스의 문맥은 PCB에 포함되며 문맥 교환시간은 오버헤드 입니다. 교환하는 동안 시스템은 유용한 일을 할 수 없고 운영체제와 PCB가 복잡할 수록 시간이 더 걸립니다. 문맥 교환 시간은 하드웨어의 지원에 좌우되는데 메모리 속도가 여러 개의 레지스터의 갱신을 하나의 명령으로 많은 문맥을 한 번에 옮길 수 있습니다. 실행 중인 프로세스에 인터럽트가 발생하면 운영체제가

다른 프로세스를 실행 상태로 바꾸고 제어를 넘겨줄 프로세스 문맥 교환이 발생합니다. 입출력 인터럽트시에는 입출력 동작이 발생한 것을 확인하고, 이벤트를 기다리는 프로세스를 준비 상태로 바꾼 뒤 실행할 프로세스를 결정합니다.
클록 인터럽트 시에는 실행 중인 프로세스 할당 시간을 초과하고 준비 상태로 바꾼 뒤 다른 프로세스를 실행 상태로 전환합니다. 인터럽트는 인터럽트 처리 루틴을 실행한 뒤 현재 실행중인 프로세스를 계속실행할 지 결정을 인터럽트가 곧 문맥교환으로 발전하지는 않습니다. 보통 이전 프로세스의 상태 레지스터 내용을 보존하고, 다른 프로세스의 레지스터에 적재하여 프로세스를 교환하는데, 이런 일련의 작업을 문맥교환이라 합니다. 아래의 그림 12.가 그 예시입니다.

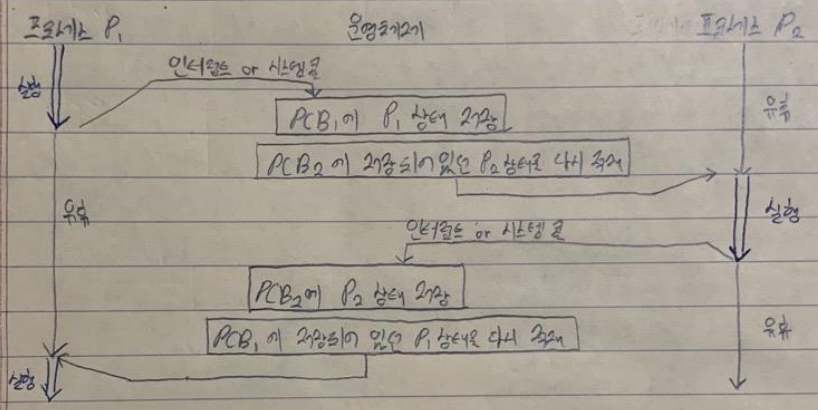


그림 12.

5. 프로세스 간 통신

협력적 프로세스들은 데이터를 교환할 수 있는, 즉 서로 데이터를 보내거나 받을 수 있는 프로세스 간 통신(interprocess communication, IPC) 기법이 필요합니다. 프로세스간 통신에는 기본적으로 공유 메모리(shared memory)와 메시지 전달(message passing)의 두 가지 모델이 있습니다. 아래의 그림 13.은 메시지 전달 모델이며 그림 14.는 공유 메모리 모델입니다.

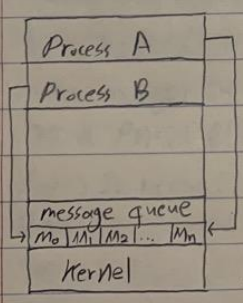


그림 13.
Message passing

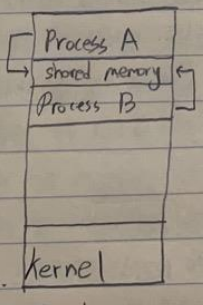


그림 14.
Shared-memory

5.1 공유 메모리 Shared Memory

공유 메모리는 통신하려는 프로세스들 사이의 공유된 메모리 영역입니다. 통신은 운영체제가 아닌 이들 프로세스의 코드에 의해 이루어집니다. 중요한 문제는 프로세스들이 공유 메모리에 접근할 때 이들 프로세스가 그 동작을 조정하고 동기화할 수 있도록 하는 메커니즘을 제공하는 것입니다. 협력하는 프로세스의 일반적인 패러다임은 생산자-소비자 문제를 생각해보겠습니다. 생산자 프로세스는 정보를 생산하고 소비자 프로세스는 그 정보를 소비합니다. 이를 위해 buffer 가 필요하며 두 가지 유형의 버퍼가 사용됩니다. 무한 버퍼(unbounded buffer)는 버퍼의 크기에 실질적인 한계가 없고 유한 버퍼(bounded buffer)는 버퍼의 크기가 고정되어 있다고 가정합니다. 유한 버퍼의 경우에는 버퍼가 비어 있으면 소비자는 반드시 대기해야 하며, 모든 버퍼가 채워져 있으면 생산자가 대기해야 합니다.

```
#define BUFFER_SIZE 10

typedef struct {
    ...
} item;

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

위와의 코드의 변수는 생산자와 소비자 프로세스가 공유하는 메모리 영역에 존재하며 공유 버퍼는 두 개의 논리 포인터 in 과 out 으로 원형 배열로 구현됩니다. 아래의 완벽 모드는 공유 메모리를 사용한 생산자 프로세스이고 오른쪽은 공유 메모리를 사용한 소비자 프로세스입니다.

```
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;
}
```

공유 메모리 사용한 생산자 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 사용한 소비자 프로세스

생산자 프로세서는 next_Produced 라는 지역 변수에 다음번 생산되는 item을 저장하고 있고 소비자 프로세서는 next_consumed 라는 지역 변수에 다음번 소비되는 item을 저장하고 있습니다. 이러한 방법은 올바르기는 하지만 최대 BUFFER_SIZE -1 까지만을 버퍼에 수용할 수 있습니다.

5.2 메시지 전달 Message Passing

메시지 전달 방식은 공유된 주소 공간을 공유하지 않고도 프로세스들이 통신을 하고, 그들의 동작을 동기화할 수 있도록 허용하는 기법을 제공합니다. 메시지 전달 시스템은 최소한 두 가지 연산을 제공합니다 send(message)와 receive(message) 입니다. 프로세스가 보낸 메시지는 고정 길이일 수도 있고 가변 길이일 수도 있습니다.

프로세스 P와 Q가 통신을 원한다면 이들 사이에는 통신 연결(communication link)이 설정되어야 하고 send() 와 receive() 를 통해 서로 메시지를 주고 받아야합니다. Communication link를 구현하는 물리적 구현(공유 메모리, 하드웨어 버스, 네트워크)과 논리적 구현이 존재하는데 여기서는 논리적 구현만 살펴보겠습니다.

5.2.1 직접 통신 간접통신

Direct Communication에서 프로세스들은 서로의 이름을 명시해야합니다.

- send(P, message) - 프로세스 P에 메시지를 전송한다.
- receive(Q, message) - 프로세스 Q로 부터 메시지를 수신한다.

이러한 방식에서 communication link는 자동으로 구축되며 두 프로세스 사이에 오직 하나의 link만 존재합니다. Link는 단방향일 수 있지만 보통 양방향입니다.

Indirect Communication에서 메시지는 메일박스 또는 포트로 송신되고, 그것으로부터 수신된다. 각 메일박스는 고유의 ID를 가집니다. 프로세스들은 오직 메일 박스를 통해서만 통신할 수 있습니다.

- send(A, message) - 메시지를 메일박스 A로 송신한다.
- receive(A, message) - 메시지를 메일박스 A로부터 수신한다.

이러한 방식에서 communication link는 프로세스들이 공유 메일 박스를 가질때만 구축되며 연결은 두 이상의 프로세스와 연관될 수 있습니다. 또한 각 프로세스 사이에는 다수의 서로 다른 연결이 존재할 수 있고 연결은 단방향이거나 양방향입니다. 간접 통신은 1. 새로운 메일 박스를 생성 2. 메일박스를 통해 메시지 송수신 3. 메일박스 삭제의 순으로 진행됩니다.

프로세스 P_1, P_2, P_3 가 모두 메일박스 A를 공유한다고 가정했을 때, P_1은 메시지를 송신하고 P_2, P_3는 메시지를 수신합니다. 이때 어느 프로세스가 P_1이 보낸 메시지를 수신하는가에 대한 답은 운영체제가 선택한 기법에 따라 달라집니다.

- 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. 시스템은 송신자에게 수신자가 누구인지 알려줄 수 있다.

5.2.2 동기화 Synchronization

메시지 전달은 blocking과 non-blocking의 방식으로 분류될 수도 있습니다. Blocking은 Synchronous 로 불 수 있습니다.

- blocking send - 송신하는 프로세스는 메시지가 수신될 때까지 봉쇄된다.
- blocking receive - 메시지가 이용가능할 때까지 수신 프로세스가 봉쇄된다.

(동시에 발생하는 즉, sender와 receiver가 동시에 실행하는 것)

Non-blocking은 Asynchronous로 볼 수 있습니다.

- non-blocking send - 송신하는 프로세스가 메시지를 보내고 작업을 재개한다.
- non-blocking receive - 송신하는 프로세스가 유효한 메시지 또는 NULL을 받는다.

send()와 receive()의 다른 조합도 가능합니다. send()와 receive()가 모두 봉쇄형일 때, 송신자와 수신자 간에 랑데뷰(rendezvous)를 하게됩니다. 이런 경우 송신자-소비자 문제에 대한 해결책은 사소한 문제가 됩니다. 아래의 코드는 이런 경우를 실행하고 있습니다.

```
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 */
}
```

5.2.3 버퍼링 Buffering

통신이 직접적이든 간접적이든 간에, 통신하는 프로세스에 의해 교환되는 메시지는 임시 큐에 들어가 있으며 이러한 큐를 구현하는데는 세가지 방식이 있습니다.

1. Zero capacity(무용량) : 큐의 길이가 0이다. 송신자는 메시지가 수신될 때까지 기다려야 한다.
2. Bounded capacity(유한용량) : 큐는 유한한 길이 n을 가진다. 큐가 가득차면 송신자는 기다려야 한다.
3. Unbounded capacity(무한용량) : 큐는 잠재적으로 무한한 길이를 가진다. 송신자는 기다리지 않는다.