1.

(1) 가상메모리에서의 Page Table은 해당 가상메모리 주소의 영역이 실제로 사용되고 있는지에 대한 여부를 가리키는 영역(valid-invalid bit)과, 만약 사용중인 영역이라면 그 영역이 매핑되는 물리적인 메모리 주소를 가리키는 영역(frame number)로 이루어져있다. 어떤 프로그램에서 메모리를 접근할 때, 페이지 테이블을 이용해 실제 메모리에 접근을 시도하는데, 작동 방식은 다음과 같다.

1. valid-invalid bit의 값을 통해 해당 메모리가 사용중인 메모리인지 확인한다.
2. 만약 사용중인 메모리라면 page table에서 frame 번호를 알아내 실제 물리 메모리에 접근할 수 있도록 한다.
3. 만약 사용중이지 않은 메모리 영역인 경우 우선 프로그램을 강제종료 시킨 후 Page Fault Exception을 발생시켜 운영체제로부터 새로운 메모리 영역을 할당받은 뒤 프로그램을 재실행한다.

(2) 프로세스에 주소공간이 할당되는 경우는 접근하려는 메모리가 invalid 상태일 때 Page Fault Exception이 발생되고 운영체제로 Trap된 것이 처리되었을 때 할당된다. 페이지 테이블이 invalid 상태인 경우 운영체제로 Page Fault Exception의 발생과 함께 Trap되고, 예외처리 핸들러가 실행된다. 운영체제는 비어있는 메모리 공간을 찾아 프로그램에서 사용될 수 있도록 보조기억장치에서 프로그램과 관련된 내용들을 물리적인 메모리로 복사해오고, 이 복사한 영역의 주소를을 페이지 테이블에 기록함으로써 Address Space가 만들어진다. 이렇게 만들어진 Address Space를 프로그램에 할당하게 되면 실제 메모리에 접근할 수 있는 상태가 되므로 Terminate되었던 프로그램이 다시 실행된다.

2.

Trashing 이란 가상 메모리 기법에서 일정 갯수 이상의 프로세스가 메모리에 load되었을 때 page fault가 빈번이 발생하는 현상을 일컫는 말이다. Trashing이 발생하는 이유는 너무 많은 프로세스가 메모리에 load 되어 충분한 여유 메모리 공간이 없을 때 각 프로세스에 할당되는 memory frame의 숫자가 감소하게 되어 충분한 공간의 메모리를 할당받지 못하기 때문이다. 이로 인해 CPU에서 page fault exception을 처리하는 횟수가 많아지고, 이로 인한 부하(overhead)가 발생하게 된다.

프로세스에 할당하는 메모리가 충분히 많을수록 page fault의 발생 횟수를 줄어들게 된다. 이를 위해 프로세스에 몇 개의 메모리가 할당되면 적당한지에 대한 측정이 필요한데, 이 과정에서 '메모리 지역성(memory locality)' 이라는 개념을 사용한다. 메모리 지역성이란, 하나의 프로세스가 실행되는 과정에서 특정 영역의 메모리를 반복적으로 접근하는 현상을 일컫는 말이다. 이 영역의 크기만큼 메모리 할당이 보장될 수 있도록 하는 것이 Trashing 문제를 해결하는 기저가 된다.

지역성을 활용한 해결 기법은 2가지 있다.

① Working-Set Model : 프로세스를 실행하는 시간 중 일부 시간동안 참조된 메모리 영역을 하나의 집합으로 묶은 후 (이 집합을 Working-Set 이라고 한다.) 이 집합의 크기만큼만 프로세스에 할당해주는 방식이다.
이 방식은 일정 시간 간격을 의미하는 delta(△) 값을 프로세스의 주기에 맞게 설정한다면 메모리 참조 갯수를 프로세스가 필요로 하는 메모리의 최적의 개수에 맞게 추출하는 것이 가능한 점에서 이론적으로 가장 좋은 방식이다.

② Page-Fault Frequency : 프로세스에 할당해 줄 메모리의 크기를 Page Fault가 얼마나 자주 발생했는지 비율로 판단하여 최적의 크기를 찾아나가는 방식이다. 할당해주는 메모리의 크기를 점진적으로 증가시켜나가면서 Page Fault가 얼마나 자주 발생했는지에 대한 비율을 측정한다. 그 후 적정 비율로 판단되는 영역 중 높은 값을 lower bound, 낮은 값을 lower bound로 설정하여 그 범위의 비율이 되도록 하는 메모리 크기만큼 프로세스에 할당해주는 방식이다.

하지만 실제로는 위의 두 가지 방법 중 Page-Fault Frequency가 가장 많이 쓰이는 방식이다. Working-Set의 크기만큼 메모리를 할당한다면 최적의 개수로 메모리를 관리하는 것이 가능하지만, 각 주기마다 Working-Set에 들어가는 메모리 개수를 확인하는데 있어서의 Runtime Overhead가 크기 때문에 비교적으로 부담이 적은 Page-Fault Frequency 방식을 이용해 일정 범위 내의 갯수만큼 메모리를 할당해주고 Page Fault 발생 비율이 높아지만 추가적으로 메모리를 할당해주는 방식을 사용한다.

3.

fork() 시스템 콜은 자식 프로세스를 생성하면서 동시에 부모 프로세스의 Code, Stack&Heap, Data와 같은 모든 것을 복사하는 명령어이다.

vfork() 시스템 콜은 전체적인 실행 방식은 fork()와 비슷하지만, 페이지의 복사가 일어나지 않고 그 부모 프로세스가 사용하던 메모리를 그대로 승계받아 사용한다. 이로 인해 fork() 시스템 콜보다 시간적인 측면에서 더 유리한 작업이다. 하지만, 이 경우 자식 프로세스에서의 데이터 변경 내용이 부모 프로세스의 데이터에도 영향을 줄 수 있다는 위험성이 존재한다.

clone() 시스템 콜은 개념적으로 thread에서의 자식 생성 방식과 비슷하다. 즉, 일부 고유 데이터 영역들은 복사하지 않으면서, code나 공통된 데이터와 같은 일부 자원을 서로 공유할 수 있는 자식을 생성한다.

4.

전세계적으로 가장 많이 쓰이고 있는 모바일 운영체제인 Android는 리눅스 커널을 기반으로 동작되는 운영체제이다. 애플리케이션은 주로 JAVA나 파생계열 언어인 Kotlin을 이용해 개발하며, JVM이 아닌 Android Runtime이라는 Android 전용 가상 머신에서 구동한다.

또한 Android는 구글에 의해 개발되고는 있지만 오픈 소스로 공개된 운영체제이기도 하다. 운영체제의 핵심인 커널부터 소프트웨어 개발 툴에 이르기까지 누구나 무료로 열어볼 수 있도록 되어있으며, 이를 이용해 독자적인 파생 안드로이드를 제작하는 것 또한 가능하다. 이러한 점은 안드로이드가 현재 전 세계적으로 가장 높은 점유율을 갖고 있는 이유 중 하나이기도 하다.

Linux 커널을 기반으로 하고 있으며 가상 머신의 개념을 이용해 돌아가는 운영체제인 만큼 대부분의 하드웨어 환경을 지원하기 때문에, 호환성이 가장 뛰어난 모바일 운영체제이다. 하지만 가상 머신의 특성상 많은 리소스를 필요하기도 하고 성능적인 측면에서는 커널 위에서 직접 동작하는 운영체제보다는 비교적으로 뒤쳐진다. 하지만 위에서 언급한 Android만의 다양한 특징으로 인해 가장 많이 쓰이는 운영체제이다.