TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2018-03-23 (22 회차)

강사: Innova Lee(이상훈)

gcccompil3r@gmail.com

학생: 정유경

ucong@naver.com

1. ls -R 구현하기　ls_module9.c (ls 를 마무리하자)

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

// 메인 앞에 함수의 프로토타입을 기술한다
void resursive_dir(char* dname);

int main(int argc, char* argv[])
{
	recursive_dir("."); // 현재 디렉토리에 대해 재귀함수 호출
	return 0;
}

void recursive_dir(char* dname) // 재귀함수 인자로 현재 디렉토리 "." 전달
{
	struct dirent* p;
	struct stat buf;
	DIR* dp; // 디렉토리포인터 dp
	chdir(dname); // 경로의 위치를 바꿔준다 // dname 을 출력하면 . 뜬다
	dp=opendir(".");
	printf("₩t %s : ₩n",dname);
	while(p=readdir(dp)) // 디렉토리 내의 파일 리스트 없을때까지 출력
		printf("%s₩n", p->d_name);
	rewinddir(dp); // 파일포인터를 다시 처음으로 이동하여 dp 를 다시 사용할 수 있도록 한다
	while(p=readdir(dp)) // 디렉토리 내의 파일 리스트 없을때까
	{
		stat(p->d_name,&buf);// 시스템 콜 stat: int stat(const char *path, struct stat *buf);
		if(S_ISDIR(buf.st_mode))
			if(strcmp(p->d_name,".") && strcmp(p->d_name, "..")) //이름이 "." ".."이 아니면
			 recursive_dir(p->d_name); // 그 (하위) 폴더에 대해 재귀함수 호출
			//if 부분 shortcut!! strcmp 는 같으면 0 반환
	}
	chdir(".."); // 하위 폴더내의 목록을 모두 출력한 후 상위로 디렉토리를 바꾼다
	closedir(dp); // recursive 호출한 곳으로 돌아간다
}
```

2. 페이징을 사용하는 이유

- 대용량의 data 를 쓰기 위함
- Swap 옵션이 붙어있지 않으면 메모리가 꽉 차기 전까지는 swap 하지 않는다.
- 페이징은 성능에는 도움이 되지 않는다. 메모리가 많이 필요할때 사용한다.
- AVR, MCU 에는 페이징이 없다.
  입력되는 데이터도 적고, 클럭 속도도 느리고 페이징을 하는 것은 비효율적이기 때문이다.
  그래서 메모리에 1:1 로 맵핑한다. 즉 다이렉트로 사용하기 때문에 디버깅 시 받는 주소값이 진짜 주소값이다.
- 페이징을 한다 = "가짜주소를 사용한다"

*. Intel CPU(CSIC)와는 달리 ARM 은 RISC 이기 때문에 소비전력이 적다.

3. fork() 하여 parent 와 child 의 PID 를 확인해보기

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main()
{
        pid_t pid; // pid_t 는 int 로 바꾸어도 지장없다
        pid=fork(); // fork();해서 pid 반환
        if(pid>0) // 자식이 있으면, 즉 부모 프로세스
                printf("parent:pid=%d,cpid=%d₩n",getpid(),pid);
        else if(pid==0) // 자식이 없으면 즉, 자식프로세스
                printf("child: pid=%d, cpid=%d₩n", getpid(),pid);
        else
        {
```

```
                perror("fork()");
                exit(-1);
        }
        return 0;
}
```

4. context switching - context.c

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main()
{
        pid_t pid;
        int i;
        pid=fork();
        if(pid >0) // 부모
        {
                while(1)
                {
                        for(i=0;i<26;i++)
                        {
                                printf("%c",i+'A'); // 대문자 순서대로 출력
                                fflush(stdout);
                        }
                }

        }

        else if(pid == 0) // 자식
        {
                while(1){
                        for(i=0;i<26;i++)
                        {
                                printf("%c",i+'a'); // 소문자 순서대로 출력
                                fflush(stdout);  //   스트림을 비운다 데이터 입출력하는 모든것들
```

```
                }
            }
        }

        else
        {
            perror("fork()");
            exit(-1);
        }
        printf("₩n");
        return 0;
}
```

5. Process 는 가상메모리를 공유할수 없다. 하지만,

하나의 프로그램을 여러 프로세스가 협력하도록 작성해야 할 경우가 생기면 이때는 오히려 각 프로세스가 독립된 메모리 공간을 갖는 제약 때문에 협력 작업이 불가능 하다

여러프로세스가 협력적으로 동작하는 프로그램을 작성할때 사용할 수있는 것이 쓰레드와 **IPC**(Inter Process Communication)이다.

*. **Pipe** 는 IPC 의 한 종류이다. 프로세스 사이에서 간단한 통신을 할때 사용되며
프로세스가 파이프의 핸들러를 공유하게 하기 위해 fork()를 사용한다.

5-1 ps_test1.c

```
#include <stdio.h>
int main()
{
        int a=10;
        printf("&a=%#p₩n",&a);
        sleep(1000);
        return 0;
}
```

5-2 ps_test2.c

```
#include <stdio.h>
int main()
{
        int *p=0xbfcb3e40;
        printf("&a=%#p₩n",p);
        return 0;
}
```

6. COW

COW 는 Copy on Write 의 약자이다.
시스템에서 fork 시에 모든 것을 생성하지 않고, 실제로 쓰기 동작이 일어날때 새로운 영역을 할당받는다. fork 시에 부모가 갖고 있는 모든 것을 복제하기에는 시스템 자원의 소모가 크고 참조만 이루어지는 공간의 경우 복제할 필요가 없기 때문에 사용하는 기법이다.
최초 fork 시에 부모로부터 상속받은 data, stack, heap 의 주소를 포함한 자료만을 복사해 오게 된다.
이후 실제 쓰기 동작이 일어날때 read-only 로 표시된 영역에 접근하기 때문에 mmu 에서 에러가 발생하고 그때 새로운 영역을 할당해주는 방식이다.
모든 영역은 부모 프로세스와 동일한 mmu table 을 가지고 있다.
실제 사용하는 영역에서만 메모리를 새롭게 할당함으로써 시스템의 성능과 효율을 높일 수 있다.

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int global =100;

int main()
{
        int local =10;
        pid_t pid;
        int i;
```

```
	pid=fork();

	if(pid>0)
	{
		global++;

		printf("global: %d, local:%d₩n",global,local);
	}

	else if(pid==0)
	{
		global++;
		local++;
		printf("global: %d, local: %d₩n", global, local);
	}

	else
	{
		perror("fork() ");
		exit(-1);
	}

	printf("₩n");
	return 0;
}
```

7. 프로세스를 분할하여 논블록킹 기능 구현하기 example2.c

parent 는 입력만 기다리고, child 는 cat > myfifo 로 리다이렉션 할 것을 기다린다.
즉, 키보드 입력과 파이프 입력이 따로 처리가 되어 논 블록킹 기능이 구현된다.

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>

int main()
{
	int fd, ret;
	char buf[1024];
```

```
        pid_t pid;
        fd=open("myfifo", O_RDWR);
        if((pid=fork()) > 0)
        {

                for(;;)
                {
                        ret = read(0,buf,sizeof(buf));
                        buf[ret]=0;
                        printf("Keyboard: %s₩n", buf);
                }
        }

        else if(pid==0)
        {

                for(;;)
                {
                        ret = read(fd,buf,sizeof(buf));
                        buf[ret]=0;
                        printf("myfifo: %s₩n",buf);
                }
        }
        else
        {
                perror("fork() ");
                exit(-1);
        }
        close(fd);
        return 0;
}
```

8. 좀비 프로세스

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>

int main(void)
{

        pid_t pid;
```

```
        if((pid=fork())>0)
                sleep(1000); // wait Q 로 빠진다. CPU 를 점유하지 않는다

        else if(pid==0)
                ;
/* 자식이 태어나자마자 종료되었다. parent 에 보고하지만 sleep 하여 알수 없다.
sleep 이 끝나면 <defunc>는 사라진다 */
        else
        {
                perror("fork() ");
                exit(-1);

        }
        return 0;
}
```

프로세스가 리눅스에서 종료될 때 그 즉시 메모리에서 제거되지 않는다. Process Descriptor 가 메모리에 남는다. (Process Descriptor 는 매우 적은양의 메모리만 차지한다) 프로세스 상태는 EXIT_ZOMBIE 가 되며 부모 프로세스에게 자식 프로세스가 SIGCHLD 신호로 종료되었음을 알린다. 그러면 부모 프로세스는 자식 프로세스의 종료 상태와 기타 정보를 읽기 위해 wait() 시스템 콜을 실행하여야 한다. 부모 프로세스는 죽은 프로세스로부터 정보를 얻는 것이 허용되어 있다. wait()이 호출된 후 좀비 프로세스는 메모리에서 완전히 제거된다.

이러한 과정은 매우 빠르게 일어나기 때문에 시스템에 좀비 프로세스에 누적되는 것을 볼 수 없을 것이다. 하지만 부모 프로세스가 제대로 프로그래밍되지 않았다면 wait()을 호출하지 않을 것이며 좀비 프로세스는 메모리에 존재할 것이다.

9. wait(&status) 시스템 콜 - wait.c

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
        pid_t pid;
        int status;
        if((pid = fork())>0)
        {
                wait(&status); // 자식의 상태를 받는 시스템 콜
                printf("status: %d₩n",status);
```

```
	}

	else if(pid==0)
		exit(7);

	else
	{
		perror("fork()");
		exit(-1);
	}
	return 0;
}

/* 실행하면 1792 가 나온다.
1792 를 8 비트 shift 하면 (1792 >> 8) 7 이 나온다.
else if(pid==0)
		exit(7); 에서의 7 의 값임을 알 수 있다. */
```

*. exit()함수

exit()함수는 프로그램을 정상종료 시키며, 이때 종료 값으로 **status** 를 부모프로세스에 status & **0377** 로넘겨준다. 부모프로세스는 wait(2)를 이용해서 자식 프로세스의 종료값을 읽어 올 수 있다.

```
#include <stdlib.h>
void exit(int status);
```

*. fork 함수로 생성된 자식 프로세스는 독립적으로 실행된다. 따라서 부모프로세스는 자식프로세스가 하는 일을 알 수 없고, 변수의 공유나 자식프로세스가 계속 살아있는지 등에 대한 정보를 기본적으로는 알 수 없다.

*. Wait() 시스템 콜

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

#include <sys/wait.h>

pid_t wait(int *status);
```

wait 시스템 콜은 주로 fork() 를 이용해서 자식 프로세스를 생성했을때 사용한다. wait() 를 쓰면 자식프로세스가 종료할때까지 해당영역에서 부모프로세스가 sleep() 모드로 기다리게 된다. 이는 자식프로세스와 부모프로세스의 동기화를 위한목적으로 부모프로세스가 자식프로세스보다 먼저 종료되어서 자식프로세스가 고아 프로세스(PPID 가 1)인 프로세스가 되는걸 방지하기 위한 목적이다. 만약 자식 프로세스가 종료되었다면 함수는 즉시 리턴되며, 자식이 사용한 모든 시스템자원을 해제한다. 그런데 어떤이유로 부모가 wait()를 호출하기 전에 자식 프로세스가 종료버리는 경우도 있다(잘못된 메모리 연산등으로 인한 죽음, 혹은 정상적으로), 이럴 경우 자식프로세스는 좀비프로세스가 되는데, wait()함수는 즉시 리턴하도록 되어있다. wait()의 인자 status 를 통하여 자식 프로세스의 상태를 받아올수 있는데, 자식프로세스의 상태값은 자식프로세스의 종료값 * 256(FF) 이다.

10. 정상종료와 비정상 종료 – wait.c

(How to extract status)

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
	pid_t pid;
	int status;
	if((pid = fork())>0)
	{
		wait(&status);
		printf("status: 0x%x₩n",(status>>8)&0xff); // 정상종료 8 비트 쉬프트하면 7 이 나올것
		// printf("status: 0x%x₩n",WEXITSTATUS(status));
		// printf("status: %d₩n",status-128);
		// printf("status: %d₩n",status&0x7f);
	}

	else if(pid==0)
		exit(7);

	else
	{
		perror("fork()");
		exit(-1);
	}
	return 0;
}
```

(추가과제) 11. ls -alR 구현 - 아직 실력이 부족하여 구현하지 못했습니다. 분발하겠습니다.