

Aufgabe 1: (17 Punkte)

Vervollständigen Sie das Programm `brownie_factory`, welches eine Simulation eines berühmten Süßigkeitenfabrikanten übernimmt.

Die Fabrik stellt für die Produktion Arbeiter ein, die die Schokoladenziegen melken, bis diese keinen Ertrag mehr bringen. Um Protesten der Belegschaft wegen Massenbeschäftigung entgegenzuwirken dürfen maximal 12 Arbeiter zeitgleich angeheuert werden. Implementieren Sie die einzelnen Arbeitsbereiche des Fabrikanten Frank, des Beobachters Jeff und der als Arbeiter eingestellten Koalas.

Per Eule (Signal) kommen zwischendurch neue Börsen- und Einkaufskurse in die Fabrik. Deren konkrete Behandlung soll hier aber nicht implementiert werden. Diese Informationen werden beispielsweise in der Funktion `buy_goat` verwendet.

Der Fabrikant Frank übernimmt die folgenden Aufgaben:

- Er initialisiert alle notwendigen Variablen und Zustände der Fabrik.
- Er lädt Jeff ein, die Fortschritte zu beobachten.
- Solange es neue Ziegen gibt, heuert er Koalas als Arbeiter an (max 12 gleichzeitig) und gibt jedem eine Ziege zum Melken.
- Er wartet darauf, dass die verbleibenden Arbeiter fertig werden.
- Er sagt Jeff Bescheid, dass seine Aufgabe beendet ist.
- Er wartet, dass Jeff seine Aufgabe beendet hat und beendet dann die Simulation.

Der Beobachter Jeff übernimmt folgende Aufgabe:

- Er wartet darauf, dass ein Koala Fortschritt gemacht hat.
- Er liest den Zustand aus.
- Er ruft den Zustand und die Beschäftigungssituation aus (`printf`).
- Wenn Frank nicht das Ende verkündet, wiederholt Jeff sein Vorgehen.
- Er macht eine letzte Zustandsmeldung und sagt Frank, dass er auch fertig ist.

Die Koalas übernehmen folgende Aufgabe:

- Solange die zugewiesene Ziege Schokolade liefert, wird diese gemolken.
- Jeder Teilertrag wird verzeichnet und Jeff gemeldet.
- Liefert die Ziege keinen Ertrag mehr, so kündigt der Koala.

Ergänzen sie den C-Code so, dass das beschriebene Verhalten erreicht wird. Hierfür müssen Sie die gegebenen Semaphoren so verwenden, dass die Arbeitsabläufe passend synchronisiert werden. Hinweise:

- Achten Sie auf eine korrekte Fehlerbehandlung der verwendeten Funktionen.
- Bei geteiltem Zustand muss stets für Konsistenz gesort werden.
- Die Ausgabe von Fehlern soll auf dem `stderr`-Kanal erfolgen.
- Ergänzen Sie das folgende Codegerüst so, dass ein vollständig übersetzbares Programm entsteht.
- `hire_koala()` und `invite_jeff()` starten jeweils neue Threads, die die Funktionen `koala()` und `jeff()` aufrufen.

<div>sem_destroy(3)</div>	<div>sem_destroy(3)</div> <div><div>NAME</div><div>sem_destroy – destroy a semaphore</div><div>SYNOPSIS</div><div>#include &lt;semaphore.h&gt; int sem_destroy(sem_t *sem);</div><div>DESCRIPTION</div><div>sem_destroy() destroys the semaphore at the address pointed to by <i>sem</i>. Destroying a semaphore that other processes or threads are currently blocked on (in <code>sem_wait(3)</code>) produces undefined behavior. Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using <code>sem_init(3)</code>.</div><div>RETURN VALUE</div><div>sem_destroy() returns 0 on success; on error, <code>-1</code> is returned, and <i>errno</i> is set to indicate the error.</div></div>	<div>sem_destroy(3)</div>
<div>sem_getvalue(3)</div>	<div>sem_getvalue(3)</div> <div><div>NAME</div><div>sem_getvalue – get the value of a semaphore</div><div>SYNOPSIS</div><div>#include &lt;semaphore.h&gt; int sem_getvalue(sem_t *sem, int *sval);</div><div>DESCRIPTION</div><div>sem_getvalue() places the current value of the semaphore pointed to <i>sem</i> into the integer pointed to by <i>sval</i>.</div><div>RETURN VALUE</div><div>sem_getvalue() returns 0 on success; on error, <code>-1</code> is returned and <i>errno</i> is set appropriately.</div></div>	<div>sem_getvalue(3)</div>
<div>sem_init(3)</div>	<div>sem_init(3)</div> <div><div>NAME</div><div>sem_init – initialize an unnamed semaphore</div><div>SYNOPSIS</div><div>#include &lt;semaphore.h&gt; int sem_init(sem_t *sem, int pshared, unsigned int value);</div><div>DESCRIPTION</div><div>sem_init() initializes the unnamed semaphore at the address pointed to by <i>sem</i>. The <i>value</i> argument specifies the initial value for the semaphore. The <i>pshared</i> argument indicates whether this semaphore is to be shared between the threads of a process (0), or between processes (1). Initializing a semaphore that has already been initialized results in undefined behavior.</div><div>RETURN VALUE</div><div>sem_init() returns 0 on success; on error, <code>-1</code> is returned, and <i>errno</i> is set appropriately.</div></div>	<div>sem_init(3)</div>

<div>sem_post(3)</div>	<div>sem_post(3)</div> <div><div>NAME</div><div>sem_post – unlock a semaphore</div><div>SYNOPSIS</div><div>#include &lt;semaphore.h&gt; int sem_post(sem_t *sem);</div><div>DESCRIPTION</div><div>sem_post() increments (unlocks) the semaphore pointed to by <i>sem</i>. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a <code>sem_wait(3)</code> call will be woken up and proceed to lock the semaphore.</div><div>RETURN VALUE</div><div>sem_post() returns 0 on success; on error, the value of the semaphore is left unchanged, <code>-1</code> is returned, and <i>errno</i> is set to indicate the error.</div></div>	<div>sem_post(3)</div>
<div>sem_wait(3)</div>	<div>sem_wait(3)</div> <div><div>NAME</div><div>sem_wait, sem_timedwait – lock a semaphore</div><div>SYNOPSIS</div><div>#include &lt;semaphore.h&gt; int sem_wait(sem_t *sem); int sem_trywait(sem_t *sem);</div><div>DESCRIPTION</div><div>sem_wait() decrements (locks) the semaphore pointed to by <i>sem</i>. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call. <code>sem_trywait()</code> is the same as <code>sem_wait()</code>, except that if the decrement cannot be immediately performed, then call returns an error (<i>errno</i> set to <code>EAGAIN</code>) instead of blocking.</div><div>RETURN VALUE</div><div>on success: 0; on error, the value of the semaphore is left unchanged, <code>-1</code> is returned, and <i>errno</i> is set to indicate the error.</div><div>ERRORS</div><div><b>EINTR</b> The call was interrupted by a signal handler <b>EINVAL</b> <i>sem</i> is not a valid semaphore. <b>EAGAIN</b> The operation could not be performed without blocking (<code>sem_trywait()</code> only).</div></div>	<div>sem_wait(3)</div>

```
#define MAX_KOALAS 12

volatile bool jeff_done = False;
sem_t available_jobs, finished_koalas, jeff_finished;
struct global_statistics{
    long long int brownie;
    sem_t mutex;
    sem_t change;
} statistics = {0};

void hire_koala(goat_t * goat);
void Frank(void);
void invite_jeff(void);

void frank(void) { //called by main()
    //init the factory
    sem_init(&available_jobs, 0, );
    sem_init(&finished_koalas, 0, );
    sem_init(&jeff_finished, 0, );
    sem_init(&statistics.mutex, 0, );
    sem_init(&statistics.change, 0, );

    // do regular stuff
    goat_t* cur_goat;
    int job_counter = 0;
    invite_jeff();
    while ((cur_goat = buy_goat()) != NULL) {
        ;
        ;
        ;
    }
    // Wait for Koalas to finish
     {
        ;
    }
    //Notify Jeff to terminate
    jeff_done = True;
    ;

    //wait for Jeff to terminate
    ;
    printf("Everything done. Bye\n");
}
```

```
void koala(goat_t *goat) {
    int brownie_output;
    while ((brownie_output = milk_goat(goat) != 0)) {
        ;
        statistics.brownie += brownie_output;
        ;
        ;
    }
    release_goat(goat);
    ;
    ;
}

void jeff(void) {
    while (!jeff_done) {
        ;
        ;
        long long int choc = statistics.brownie;
        ;
        int koala_slots;
        ;
        printf("brownie = %lld, workers = %d\n", choc, MAX_WORKERS - worker_slots);
    }
    printf("Well done, we produced %lld units.\n", statistics.brownie);
    ;
}
```

Annahme: Sie verwenden Prozesse statt der Threads. Nennen Sie einen daraus resultierenden Unterschied in der Interaktion und dessen Ursache.

-----

-----

-----

-----

Aufgabe 2: (11 Punkte)

2.1: Teilinterpretation: Wann wird das Betriebssystem aktiv? Nenne ein Beispiel für eine zu interpretierende Instruktion.

2.2: Worin unterscheiden sich die Behandlung der folgenden Trap-Ursachen: (a) Die logische Adresse zeigt auf eine in den Hintergrundspeicher ausgelagerte Seite; (b) Die Adresse ist nicht Teil des logischen Adressraums.

2.3: Das folgende Programmstück wird in einem Prozess eines UNIX-Systems ausgeführt:

```
// ...
int *p = malloc(sizeof(int));
*p = -1;
// ...
```

2.3.1: Welcher Fehler kann dabei auf Ebene E<sub>2</sub> auftreten? (1 Punkt)

2.3.2: Benennen Sie die Hardwarekomponente, die diesen Fehler detektiert. (1 Punkt)

2.3.3: Wie aktiviert diese Komponente das Betriebssystem? (1 Punkt)

2.3.4: Was wird das Betriebssystem mit dem Prozess machen, der den Fehler hervorruft? (1 Punkt)

2.4: Worin unterscheiden sich Online- und Offlinescheduling? Nennen Sie je einen Vorteil.

Aufgabe 3: (10 Punkte)

3.1: Interprozesskommunikation kann sowohl mittels gemeinsamem Speicher (shared memory) als auch mittels Nachrichtenaustausch (message passing) implementiert werden. Beantworten Sie dazu die folgenden gegenüberstellenden Fragen:

3.1.1: Wie wird dabei die Anwendungslogik der Kommunikationspartner jeweils synchronisiert? (2 Punkte)

3.1.2: Welche Variante eignet sich besser, um Ortstransparenz herzustellen? Begründen Sie ihre Antwort! (2 Punkte)

3.2: Sie haben in der Vorlesung verschiedene Semantiken für nachrichtenbasierte Interprozesskommunikation (IPC) kennen gelernt. Ordnen sie den folgenden Echtweltvorgängen ihre entsprechende Nachrichtensemantik zu:

Frage in einem Forum stellen	<div></div>
Kuchen beim Konditor kaufen	<div></div>
„Geöffnet“-Schild an die Ladentür hängen	<div></div>
Durchsage: „Nächster Halt...“	<div></div>
Frage stellen: „Wollen wir gleich ins Kino gehen?“	<div></div>
Abschlussarbeit zur Bewertung in Briefkasten einwerfen	<div></div>

**Aufgabe 4: (34 Punkte)**

Auf einem Rechensystem befinden sich diverse Dateien und Verzeichnisse in einer Verzeichniss-  
struktur und das Programm `inspect` um aus diesen Informationen zu gewinnen. Zudem ist das  
Programm `collect` vorhanden, welches die Ausgaben mehrerer `inspect`-Ausführungen bündeln  
kann. Schreiben Sie das Programm `combinator` welches eine Instanz des Programms `collect`  
mit dem Parameter `OUT_FILE` startet. Für jeden Kommandozeilenparameter `IN_FILE` startet  
`collect` eine Instanz des Programms `inspect`. Die Ausgaben der `inspect`-Prozesse sollen als  
Eingabe an den `collect`-Prozess gegeben werden. Die Ausgaben des `collect`-Prozesses sollen  
in die Datei `OUT_FILE` geschrieben werden.

Aufrufsyntax: `combinator OUT_FILE IN_FILE...`

Implementieren Sie das Programm `combinator`, welches mit einer Liste von Pfaden aufgerufen  
wird, die den jeweiligen zu startenden Programmen übergeben werden.

Allgemeine Anforderungen:

- Tritt ein nicht erwarteter Fehler auf, so ist der Prozess mit Ausgabe einer Fehlermeldung zu  
beenden. Hierzu steht die Funktion `die(char *msg)` bereit.
- Alle unnötigen Dateideskriptoren sind an geeigneter Stelle zu schließen (siehe `close_fds`).

Funktion: `int main(int argc, char *argv[])`

- Prüfen der Anzahl der Parameter
- Nutzungsausgabe auf der Fehlerausgabe, wenn die Anzahl der Parameter falsch ist
- Das Programm `collect` mittels der Funktionen `start` starten.
- Für jeden gegebenen Pfad das Programm `inspect` mittels der Funktion `start` starten.
- Auf das Beenden aller Kindprozesse warten

Funktion: `int start(char *prog, char *f_name, int pipe_ends[],  
void prep_child(char *, int []))`

- Starten eines neuen Kindprozesses
- Vorbereiten des Kindzustandes mittels des gegebenen Funktionszeigers (`prep_child`).
- Ausführen des Programms `prog` im Kindprozess
- Rückgabe: Prozess-ID des Kindes

Funktion: `void prep_inspect(char *f_name, int pipe_ends[])`

- Setzen der Ausgabe auf die gegebene Pipe

Funktion: `void prep_collect(char *f_name, int pipe_ends[])`

- Erzeugen oder Öffnen der Datei `f_name` zum Überschreiben. Zugriffsrechte: Nur Eigentümer  
darf lesen und schreiben.
- Setzen der Ausgabe auf die soeben geöffnete Datei.
- Setzen der Eingabe auf die gegebene Pipe.

Funktion: `void close_fds(int fds[], int n)`

- Schließen aller gegebenen Dateideskriptoren

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

```
void die(const char *msg) {
    perror(msg);
    exit(1);
}
```

```
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

```
// Zu implementieren
int start(char *prog, char *f_name, int pipe_ends[],
          void prep_child(char *, int []));
void prep_inspect(char *f_name, int pipe_ends[]);
void prep_collect(char *f_name, int pipe_ends[]);
void close_fds(int fds[], int n);
```

```
int main(int argc, char *argv[]) {
```

**S:**

**C:**

**IV.**

```
int start(char *prog, char *f_name, int pipe_ends[],
```

```
void prep_child(char *, int [])) {
```

```
void close_fds(int fds[], int n) {
```

```
void prep_collect(char* f_name, int pipe_ends[]) {
```

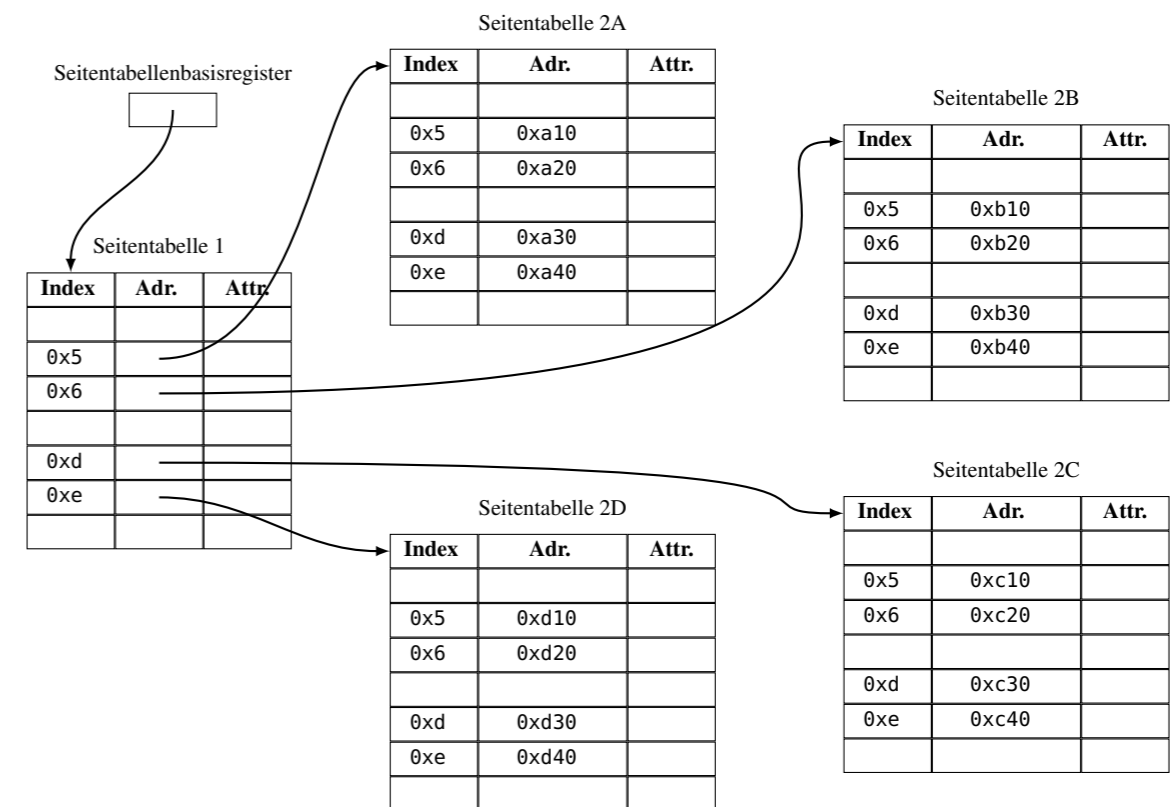
**PC:**

```
void prep_inspect(char *fname, int pipe_ends[]) {
```

**PI:**

### Aufgabe 5: (13 Punkte)

Auf einem Mikrocontroller mit byteweise adressiertem Speicher ist seitenorientierter logischer Adressraum mit zweistufiger Hierarchie implementiert. Die Adressen sind 12 Bit breit. Pro Stufe werden 4 Bit zur Indizierung verwendet. Die verbleibenden Bits werden als Offset in die Seite verwendet. Es sind außerdem 3 Bit für Attribute im Seitendeskriptor vorgesehen. Die Wortbreite beträgt 2 Byte. Gegeben sei unten dargestellte Hierarchie zweistufiger Seitentabellen.



**5.1:** Bestimmen sie die **reale Adresse** zur logischen Adresse 0xed6. Geben Sie hierbei die zur Bestimmung notwendigen Zwischenschritte stichpunktartig an!

**5.2:** Nehmen Sie an, dass alle gültigen Seitentableneinträge oben abgebildet sind. Was würde in diesem Fall mit einem Prozess passieren, der schreibend auf die Adresse 0x799 zugreift und wieso?

- 12 von 17 -

**5.3:** Bestimmen Sie die folgenden Größen:

- Mindestanzahl Seitentabellen für einen Prozess mit ausführbarem Code und nicht ausführbaren Daten und nur lesbaren Konstanten
- Größe einer Seite
- maximale Größe des logischen Adressraums

**5.4:** Alternativ zu der gegebenen zweistufigen Implementierung könnte man auch eine einstufige Implementierung mit 8 Bit Seitennummern verwenden. Beschreiben sie einen Vor- und einen Nachteil dieser Alternative.

**5.5:** Welchen alternativen Mechanismus gibt es zur Implementierung von virtuellem Speicher? Benennen Sie diesen und einen Vor oder Nachteil gegenüber der seitenorientierten Implementierung!

**Aufgabe 6: (5 Punkte)**

Gegeben sei ein Dateisystem mit indizierter Speicherung. Jeder Indexknoten enthält 5 direkte Verweise, und je einen einfach, zweifach und dreifach indirekten Verweis. Eine Adresse ist 7 Byte groß, ein Block 5 KiByte.

**6.1:** Wieviele Blöcke werden benötigt, um eine Datei der Größe 21 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

**6.2:** Wieviele Blöcke werden benötigt, um eine Datei der Größe 50 KiByte darzustellen? Wie werden die Datenblöcke adressiert?

**6.3:** Wie groß kann eine Datei maximal in diesem Dateisystem sein, wenn nur die direkten Verweise genutzt werden dürfen?

fork(2)

fork(2)

**NAME** fork – create a child process

**SYNOPSIS**

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

pid_t fork(void);
```

**DESCRIPTION**

**fork()** creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process is an exact duplicate of the parent process except for the following points:

- \* The child has its own unique process ID.
- \* The child's parent process ID is the same as the parent's process ID.

**RETURN VALUE**

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, `–1` is returned in the parent, no child process is created, and *errno* is set appropriately.

dup(2)

**NAME** dup, dup2 – duplicate a file descriptor

**SYNOPSIS**

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
int dup3(int oldfd, int newfd, int flags);
```

**DESCRIPTION**

The **dup()** system call duplicates a file descriptor.

The **dup2()** system call replaces the file descriptor specified in *newfd* with a copy of *oldfd*. If the file descriptor *newfd* was previously open, it is silently closed before being reused. The steps of closing and reusing the file descriptor *newfd* are performed *atomically*.

The **dup3()** is the same as **dup2()**, except that the **O\_CLOEXEC** flag can be specified in *flags*. See the description of the same flag in **open(2)** for reasons why this may be useful.

**RETURN VALUE**

On success: the new file descriptor. On error, `–1` is returned, and *errno* is set appropriately.

dup(2)

open(2)

**NAME** open, creat – open and possibly create a file

**SYNOPSIS**

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

**DESCRIPTION**

The **open()** system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O\_CREAT** is specified in *flags*) be created by **open()**.

The argument *flags* must include one of the following *access modes*: **O\_RDONLY**, **O\_WRONLY**, or **O\_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more of the following flags can be bitwise-*or*'d in *flags*:

- O\_APPEND** The file is opened in append mode.
- O\_TRUNC** The file is opened in overwrite mode. Existing content is deleted.
- O\_CREAT** If *pathname* does not exist, create it as a regular file.
- O\_CLOEXEC** Automatically close this file descriptor on **exec(2)**.

The owner (user ID) of the new file is set to the effective user ID of the process.

The *mode* argument specifies the file mode bits be applied when a new file is created. This argument must be supplied when **O\_CREAT** is specified in *flags*; otherwise *mode* is ignored.

A call to **creat()** is equivalent to calling **open()** with *flags* equal to **O\_CREAT|O\_WRONLY|O\_TRUNC**.

**RETURN VALUE**

on success: return the new file descriptor; on error: return `–1`, *errno* is set appropriately.

exec(3)

**NAME** execl, execlp, execv, execvp – execute a file

**SYNOPSIS**

```
#include <unistd.h>

int execl(const char *pathname, const char *arg, ..., NULL *);
int execlp(const char *file, const char *arg, ..., NULL *);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

**DESCRIPTION**

The **exec()** family of functions replaces the current process image with a new process image.

The initial argument for these functions is the name of a file that is to be executed.

The functions can be grouped based on the letters following the "exec" prefix.

**l - execl(), execlp()**

The *const char \*arg* and subsequent ellipses can be thought of as *arg0, arg1, ..., argn*. The list of arguments *must* be terminated by a null pointer.

By contrast with the 'l' functions, the 'v' functions (below) specify the command-line arguments of the executed program as a vector.

**v - execv(), execvp()**

The *char \*const argv[]* argument is an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a null pointer.

**p - execlp(), execvp()**

These functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character.

**RETURN VALUE**

The **exec()** functions return only if an error has occurred. The return value is `–1`, and *errno* is set to indicate the error.

exec(3)

pipe(2)	pipe(2)	
<div><div>NAME</div><div>pipe, pipe2 – create pipe</div></div> <div><div>SYNOPSIS</div><div><pre>#include &lt;unistd.h&gt; int pipe(int pipefd[2]); int pipe2(int pipefd[2], int flags);</pre></div></div> <div><div>DESCRIPTION</div><div><p><b>pipe()</b> creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array <i>pipefd</i> is used to return two file descriptors referring to the ends of the pipe. <i>pipefd[0]</i> refers to the read end of the pipe. <i>pipefd[1]</i> refers to the write end of the pipe.</p><p>If <i>flags</i> is 0, then <b>pipe2()</b> and <b>pipe()</b> are the same. The following values can be bitwise ORed in <i>flags</i>:</p><p><b>O_CLOEXEC</b> Enable the close-on-exec flag for the new file descriptor.</p><p>On success, zero is returned. On error, <b>-1</b> is returned, and <i>errno</i> is set appropriately.</p></div></div> <div><div>RETURN VALUE</div><div></div></div>		
sigaction(2)	sigaction(2)	
<div><div>NAME</div><div>sigaction – examine and change a signal action</div></div> <div><div>SYNOPSIS</div><div><pre>#include &lt;signal.h&gt; int sigaction(int signum, const struct sigaction *act,               struct sigaction *oldact);</pre></div></div> <div><div>DESCRIPTION</div><div><p>The <b>sigaction()</b> system call is used to change the action taken by a process on receipt of the signal <i>signum</i>. If <i>act</i> is non-NULL, the new action for signal <i>signum</i> is installed from <i>act</i>. If <i>oldact</i> is non-NULL, the previous action is saved in <i>oldact</i>.</p><p>The <i>sigaction</i> structure is defined as follows:</p></div></div> <div><div>RETURN VALUE</div><div><p><b>sigaction()</b> returns 0 on success; on error, <b>-1</b> is returned, and <i>errno</i> is appropriately</p></div></div>		

wait(2)	wait(2)	
<div><div>NAME</div><div>wait, waitpid – wait for process to change state</div></div> <div><div>SYNOPSIS</div><div><pre>#include &lt;sys/types.h&gt; #include &lt;sys/wait.h&gt; pid_t wait(int *wstatus); pid_t waitpid(pid_t pid, int *wstatus, int options);</pre></div></div> <div><div>DESCRIPTION</div><div><p>All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).</p><p><b>wait()</b> and <b>waitpid()</b></p><p>The <b>wait()</b> system call suspends execution of the calling thread until one of its children terminates. The call <i>wait(&amp;wstatus)</i> is equivalent to:</p><pre>waitpid(-1, &amp;wstatus, 0);</pre><p>The <b>waitpid()</b> system call suspends execution of the calling thread until a child specified by <i>pid</i> argument has changed state. By default, <b>waitpid()</b> waits only for terminated children, but this behavior is modifiable via the <i>options</i> argument, as described below.</p><p>The value of <i>pid</i> can be:</p><ul style="list-style-type: none"><li><b>-1</b> meaning wait for any child process.</li><li><b>&gt; 0</b> meaning wait for the child whose process ID is equal to the value of <i>pid</i>.</li></ul><p>The value of <i>options</i> is an OR of zero or more of the following constants:</p><p><b>WNOHANG</b> return immediately if no child has exited.</p><p><b>WUNTRACED</b></p><p>also return if a child has stopped (but not traced via <b>ptrace(2)</b>). Status for <i>traced</i> children which have stopped is provided even if this option is not specified.</p><p><b>WCONTINUED</b> (since Linux 2.6.10)</p><p>also return if a stopped child has been resumed by delivery of <b>SIGCONT</b>.</p><p>If <i>wstatus</i> is not NULL, <b>wait()</b> and <b>waitpid()</b> store status information in the <i>int</i> to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in <b>wait()</b> and <b>waitpid()</b>):</p><p><b>WIFEXITED</b>(<i>wstatus</i>)</p><p>returns true if the child terminated normally, that is, by calling <b>exit(3)</b> or <b>_exit(2)</b>, or by returning from <b>main()</b>.</p><p><b>WEXITSTATUS</b>(<i>wstatus</i>)</p><p>returns the exit status of the child. This consists of the least significant 8 bits of the <i>status</i> argument that the child specified in a call to <b>exit(3)</b> or <b>_exit(2)</b> or as the argument for a return statement in <b>main()</b>. This macro should be employed only if <b>WIFEXITED</b> returned true.</p><p><b>RETURN VALUE</b></p><p><b>wait()</b>: on success, returns the process ID of the terminated child; on error, <b>-1</b> is returned. If no unwaited-for children exist, <b>-1</b> is returned and <i>errno</i> is set to <b>ECHILD</b> .</p><p><b>waitpid()</b>: on success, returns the process ID of the child whose state has changed; if <b>WNOHANG</b> was specified and one or more child(ren) specified by <i>pid</i> exist, but have not yet changed state, then 0 is returned. On error, <b>-1</b> is returned.</p><p>Each of these calls sets <i>errno</i> to an appropriate value in the case of an error.</p></div></div>		