1 Prozesse und Scheduling (9 Punkte)

a) Round Robin (6 Punkte) Die Prozesse P1, P2 und P3 treffen in dieser Reihenfolge im System ein und sind alle zum Zeitpunkt t=0 rechenbereit. Alle relevanten Daten zu diesen Prozessen sind in der folgenden Tabelle angegeben:

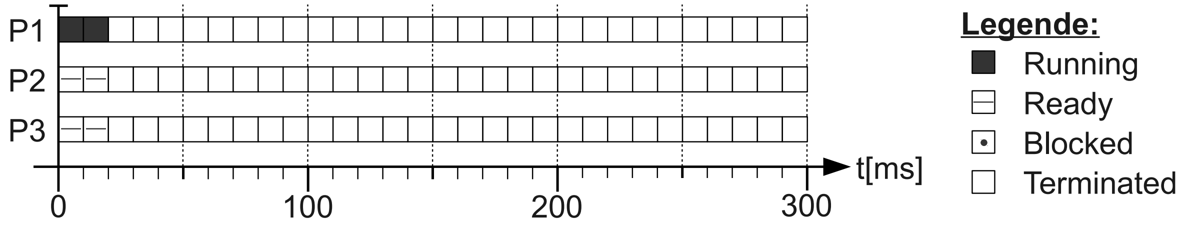
Prozess	*P1*	*P2*	*P3*
Bedienzeit	70 ms	110 ms	90 ms
E/A-Zeitpunkt	20 ms	30 ms	60 ms
E/A-Zeitdauer	80 ms	30 ms	50 ms

Bei der *Bedienzeit* handelt es sich um die reine Rechenzeit. Hinzu kommt die *Zeitdauer* der E/A-Operationen (hier blockiert der Prozess). Die E/A-Operation startet nachdem der Prozess die durch *E/A-Zeitpunkt* angegebene Zeit gerechnet hat.

Nemen Sie für die Ablaufplanung die ***Round Robin*-Strategie** mit einer **40ms Zeitscheibe** an.

Zeichnen Sie in das folgende Gantt-Diagramm ein, wie die drei Prozesse P1, P2 und P3 abgearbeitet werden. Jeder Prozess führt *genau einen* E/A-Vorgang durch. Die Prozessumschaltzeit kann vernachlässigt werden. Markieren Sie in dem folgenden Diagramm die Prozesszustände entsprechend der Legende.

Hinweis: Die ersten 20 ms sind bereits fertig ausgefüllt.



b) Allgemeine Fragen (insgesamt 3 Punkte)

1. **„>“ Shell-Operator (1,5 Punkte)** Angenommen der Befehl `echo Parameter` gibt in die Standardausgabe den Text aus dem übergebenen Parameter aus:
   Was bewirkt der Befehl `echo "" > Datei`, falls `Datei` schon einen Inhalt enthält?

2. **Zombie-Prozesse (1,5 Punkte)** Warum erbt das Betriebssystem Zombie-Prozesse (bis zu deren Abfrage) anstatt diese direkt aus der Prozesstabelle zu entfernen?

2 Synchronisation und Verklemmungen (11 Punkte)

a) Erzeuger-/Verbraucher-Problem (7 Punkte) Im Folgenden soll die Erzeuger-Funktion `producer()` Elemente mittels `produce_element()` erzeugen und durch `enqueue()` in eine gemeinsam genutzte, beliebig große Warteschlange einfügen. Eine Verbraucher-Funktion `consumer()` soll vorhandene Elemente mit `dequeue()` aus der Warteschlange nehmen und diese durch `consume_element()` verbrauchen.

Der Verbraucher soll nur dann ein Element aus der Warteschlange nehmen, wenn eines verfügbar ist und müssen daher mittels **einseitiger Synchronisation** koordiniert werden.

Die beiden Funktionen `enqueue()` und `dequeue()` werden potentiell nebenläufig ausgeführt und müssen daher mittels **gegenseitigem Ausschluss** synchronisiert werden.

Legen Sie dazu zwei geeignet **benannte Semaphore** an, initialisieren Sie diese mit einen passenden Wert. Setzen Sie anschließend an den richtigen Stellen im Code die Semaphor-Operationen `P(sem_name)` und `V(sem_name)` ein. Die Funktionen `P()`, `V()`, `produce_element()`, `consume_element()`, `enqueue()` und `dequeue()` können als gegeben angesehen werden und müssen *nicht* implementiert werden! Alle Funktionen dürfen ohne Fehlerbehandlung verwendet werden.

(Hinweis: Geläufige Synonyme für die `P()` und `V()` Operationen sind `wait()` und `signal()` bzw. `sem_wait()` und `sem_post()`.)

Namen und Initialwerte der Semaphoren: ☐ = ☐
☐ = ☐

```
producer() {
    while (1) {
        Element e = produce_element();



        enqueue(e);






   }
}

consumer() {
    while (1) {






        Element e = dequeue();



        consume_element(e);
    }
}
```

b) **Verklemmungen (4 Punkte)** Wenn eine geschlossene Kette wechselseitig wartender Prozesse existiert (*circular wait*, Zyklus im Betriebsmittelbelegungsgraphen), liegt eine Verklemmung vor.

Nennen und **erklären** Sie kurz/stichpunktartig **zwei der drei *Vorbedingungen***, die erfüllt sein müssen, damit es überhaupt zu einer Verklemmung kommen kann.

3 Speicherverwaltung und Virtueller-Speicher (12 Punkte)

a) **Platzierungs- & Ersetzungsstrategie (4 Punkte)** Erläutern Sie den Unterschied zwischen der Platzierungsstrategie (*placement policy*) und der Ersetzungsstrategie (*replacement policy*)!

b) **Speichersegmentierung (4 Punkte)** Geben Sie für die logischen Adressen `0x1000 A100` und `0x030B 5000` die zugehörigen physikalischen Adressen unter Anwendung des Speichersegmentierungsverfahrens an. Die höchstwertigen 8 Bit der logischen Adresse geben die Position innerhalb der Segmenttabelle an. Löst eine Speicheranfrage eine Zugriffsverletzung aus, so machen Sie dies bitte kenntlich.

Segmenttabelle:

	Startadresse	Länge
01$_{16}$	B542 0000$_{16}$	01 0000$_{16}$
02$_{16}$	C471 0000$_{16}$	00 F000$_{16}$
03$_{16}$	B080 0000$_{16}$	00 FFFF$_{16}$
...		
10$_{16}$	4310 1000$_{16}$	FF FFFF$_{16}$

logische Adresse: `0x1000 A100`$_{16}$
→ physikalische Adresse:

logische Adresse: `0x030B 5000`$_{16}$
→ physikalische Adresse:

c) **Buddy-Verfahren (4 Punkte)** Im Folgenden sind vier unterschiedliche Szenarien für einen teilweise belegten 32 MiB Speicher gegeben. Die zweite Zeile gibt jeweils die aktuelle Belegung an. Ergänzen Sie in jedem Szenario die neue Anforderung nach dem *Buddy*-Verfahren zur dynamischen Speicherverwaltung. Markieren Sie hierfür die belegten Speicherbereiche mit dem Prozessnamen.
Hinweis: Falls eine Belegung/Freigabe *nicht* erfüllt werden kann, kennzeichnen Sie das betreffende Szenario geeignet.

Szenario 1: Prozess C belegt 3 MiB

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
A	A	A	A							B	B				

Szenario 2: Prozess D belegt 12 MiB

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
A	A														

Szenario 3: Prozess E belegt 14 MiB

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
		B	B									A	A		

Szenario 4: Prozess F belegt 7 MiB

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
A	A	A	A	B	B										

4 Ein-/Ausgabe und Dateisysteme (7,5 Punkte)

a) **Block-Buffer-Cache (3 Punkte)** **Nennen** und **erläutern** Sie drei Ereignisse, die das Rückschreiben des Block-Buffer-Caches auslösen.

b) **I/O-Scheduling (4,5 Punkte)** Gegeben sei ein Plattenspeicher mit 8 Spuren. Der dazugehörige I/O-Scheduler bekommt immer wieder Leseaufträge für eine bestimmte Spur. Die Leseaufträge in L_1 sind dem I/O-Scheduler bereits bekannt. Nach drei bearbeiteten Aufträgen erhält er die Aufträge in L_2. Nach weiteren drei (d.h. nach insgesamt 6) bearbeiteten Aufträgen erhält er die Aufträge in L_3. Zu Beginn befinde sich der Schreib-/Lesekopf über Spur 0.
$L_1 = \{1, 4, 7, 2\}, L_2 = \{3, 6, 0\}, L_3 = \{5, 2\}$
Der I/O-Scheduler arbeitet nach der **Fahrstuhlstrategie** (Elevator). Bitte tragen Sie die Reihenfolge der gelesenen Spuren ein.

5 Programmieraufgabe (5,5 Punkte)

Implementieren Sie ein Programm `hellofork`, das einen Kindprozess erzeugt welcher `Hello world!` ausgibt. Der Elternprozess soll auf die Terminierung des Kindprozesses warten und anschließend `Ende` ausgeben. Der Kindprozess soll immer mit dem *exit*-Code 2 terminieren.

Relevante Manual-Seiten im Anhang: fork, wait

Hinweis: Einfache syntaktische Fehler (z.B. vergessene Strichpunkte) führen nicht zu Punktabzug, es geht um die sematische Umsetzung.

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

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








}
```

NAME

fork − create a child process

SYNOPSIS

#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 and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content. Memory writes, file mappings (**mmap**(2)), and unmappings (**munmap**(2)) performed by one of the processes do not affect the other.

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

- The child has its own unique process ID, and this PID does not match the ID of any existing process group (**setpgid**(2)) or session.
- The child's parent process ID is the same as the parent's process ID.
- The child does not inherit its parent's memory locks (**mlock**(2), **mlockall**(2)).
- Process resource utilizations (**getrusage**(2)) and CPU time counters (**times**(2)) are reset to zero in the child.
- The child's set of pending signals is initially empty (**sigpending**(2)).
- The child does not inherit semaphore adjustments from its parent (**semop**(2)).
- The child does not inherit process-associated record locks from its parent (**fcntl**(2)). (On the other hand, it does inherit **fcntl**(2) open file description locks and **flock**(2) locks from its parent.)
- The child does not inherit timers from its parent (**setitimer**(2), **alarm**(2), **timer_create**(2)).
- The child does not inherit outstanding asynchronous I/O operations from its parent (**aio_read**(3), **aio_write**(3)), nor does it inherit any asynchronous I/O contexts from its parent (see **io_setup**(2)).

The process attributes in the preceding list are all specified in POSIX.1. The parent and child also differ with respect to the following Linux-specific process attributes:

- The child does not inherit directory change notifications (dnotify) from its parent (see the description of **F_NOTIFY** in **fcntl**(2)).
- The **prctl**(2) **PR_SET_PDEATHSIG** setting is reset so that the child does not receive a signal when its parent terminates.
- The default timer slack value is set to the parent's current timer slack value. See the description of **PR_SET_TIMERSLACK** in **prctl**(2).
- Memory mappings that have been marked with the **madvise**(2) **MADV_DONTFORK** flag are not inherited across a **fork**().
- Memory in address ranges that have been marked with the **madvise**(2) **MADV_WIPEONFORK** flag is zeroed in the child after a **fork**(). (The **MADV_WIPEONFORK** setting remains in place for those address ranges in the child.)
- The termination signal of the child is always **SIGCHLD** (see **clone**(2)).
- The port access permission bits set by **ioperm**(2) are not inherited by the child; the child must turn on any bits that it requires using **ioperm**(2).

Note the following further points:

- The child process is created with a single thread—the one that called **fork**(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of **pthread_atfork**(3) may be helpful for dealing with problems that this can cause.

- After a **fork**() in a multithreaded program, the child can safely call only async-signal-safe functions (see **signal−safety**(7)) until such time as it calls **execve**(2).
- The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see **open**(2)) as the corresponding file descriptor in the parent. This means that the two file descriptors share open file status flags, file offset, and signal-driven I/O attributes (see the description of **F_SETOWN** and **F_SETSIG** in **fcntl**(2)).
- The child inherits copies of the parent's set of open message queue descriptors (see **mq_overview**(7)). Each file descriptor in the child refers to the same open message queue description as the corresponding file descriptor in the parent. This means that the two file descriptors share the same flags (*mq_flags*).
- The child inherits copies of the parent's set of open directory streams (see **opendir**(3)). POSIX.1 says that the corresponding directory streams in the parent and child *may* share the directory stream positioning; on Linux/glibc they do not.

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 to indicate the error.

ERRORS

EAGAIN

A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

- the **RLIMIT_NPROC** soft resource limit (set via **setrlimit**(2)), which limits the number of processes and threads for a real user ID, was reached;
- the kernel's system-wide limit on the number of processes and threads, */proc/sys/kernel/threads−max*, was reached (see **proc**(5));
- the maximum number of PIDs, */proc/sys/kernel/pid_max*, was reached (see **proc**(5)); or
- the PID limit (*pids.max*) imposed by the cgroup "process number" (PIDs) controller was reached.

EAGAIN

The caller is operating under the **SCHED_DEADLINE** scheduling policy and does not have the reset-on-fork flag set. See **sched**(7).

ENOMEM

fork() failed to allocate the necessary kernel structures because memory is tight.

ENOMEM

An attempt was made to create a child process in a PID namespace whose "init" process has terminated. See **pid_namespaces**(7).

ENOSYS

fork() is not supported on this platform (for example, hardware without a Memory-Management Unit).

ERESTARTNOINTR (since Linux 2.6.17)

System call was interrupted by a signal and will be restarted. (This can be seen only during a trace.)

CONFORMING TO

POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

NOTES

Under Linux, **fork**() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

C library/kernel differences

Since version 2.3.3, rather than invoking the kernel's **fork**() system call, the glibc **fork**() wrapper that is provided as part of the NPTL threading implementation invokes **clone**(2) with flags that provide the same effect as the traditional system call. (A call to **fork**() is equivalent to a call to **clone**(2) specifying *flags* as just **SIGCHLD**.) The glibc wrapper invokes any fork handlers that have been established using **pthread_atfork**(3).

EXAMPLES

See **pipe**(2) and **wait**(2).

SEE ALSO

clone(2), **execve**(2), **exit**(2), **setrlimit**(2), **unshare**(2), **vfork**(2), **wait**(2), **daemon**(3), **pthread_atfork**(3), **capabilities**(7), **credentials**(7)

COLOPHON

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.

NAME

wait, waitpid, waitid − wait for process to change state

SYNOPSIS

#include <sys/wait.h>

pid_t wait(int ***_wstatus_);**
pid_t waitpid(pid_t _pid_**, int ***_wstatus_**, int** _options_**);**

int waitid(idtype_t _idtype_**, id_t** _id_**, siginfo_t ***_infop_**, int** _options_**);**
/* This is the glibc and POSIX interface; see
NOTES for information on the raw system call. */

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

waitid():
Since glibc 2.26:
_XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200809L
Glibc 2.25 and earlier:
_XOPEN_SOURCE
|| /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
|| /* Glibc <= 2.19: */ _BSD_SOURCE

DESCRIPTION

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. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. 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).

If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of **sigaction**(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

wait() and waitpid()

The **wait**() system call suspends execution of the calling thread until one of its children terminates. The call *wait(&wstatus)* is equivalent to:

```
waitpid(−1, &wstatus, 0);
```

The **waitpid**() system call suspends execution of the calling thread until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

< −1	meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
−1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process at the time of the call to **waitpid**().
> 0	meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG
return immediately if no child has exited.

WUNTRACED
also return if a child has stopped (but not traced via **ptrace**(2)). Status for *traced* children which have stopped is provided even if this option is not specified.

WCONTINUED (since Linux 2.6.10)
also return if a stopped child has been resumed by delivery of **SIGCONT**.

(For Linux-only options, see below.)

If *wstatus* is not NULL, **wait**() and **waitpid**() store status information in the *int* 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 **wait**() and **waitpid**()!):

WIFEXITED(_wstatus_**)**
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(_wstatus_**)**
returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to **exit**(3) or **_exit**(2) or as the argument for a return statement in main(). This macro should be employed only if **WIFEXITED** returned true.

WIFSIGNALED(_wstatus_**)**
returns true if the child process was terminated by a signal.

WTERMSIG(_wstatus_**)**
returns the number of the signal that caused the child process to terminate. This macro should be employed only if **WIFSIGNALED** returned true.

WCOREDUMP(_wstatus_**)**
returns true if the child produced a core dump (see **core**(5)). This macro should be employed only if **WIFSIGNALED** returned true.

This macro is not specified in POSIX.1-2001 and is not available on some UNIX implementations (e.g., AIX, SunOS). Therefore, enclose its use inside *#ifdef WCOREDUMP ... #endif*.

WIFSTOPPED(_wstatus_**)**
returns true if the child process was stopped by delivery of a signal; this is possible only if the call was done using **WUNTRACED** or when the child is being traced (see **ptrace**(2)).

WSTOPSIG(_wstatus_**)**
returns the number of the signal which caused the child to stop. This macro should be employed only if **WIFSTOPPED** returned true.

WIFCONTINUED(_wstatus_**)**
(since Linux 2.6.10) returns true if the child process was resumed by delivery of **SIGCONT**.

waitid()

The **waitid**() system call (available since Linux 2.6.9) provides more precise control over which child state changes to wait for.

The *idtype* and *id* arguments select the child(ren) to wait for, as follows:

idtype == **P_PID**
Wait for the child whose process ID matches *id*.

idtype == **P_PIDFD** (since Linux 5.4)
Wait for the child referred to by the PID file descriptor specified in *id*. (See **pidfd_open**(2) for further information on PID file descriptors.)

idtype == **P_PGID**
Wait for any child whose process group ID matches *id*. Since Linux 5.4, if *id* is zero, then wait for any child that is in the same process group as the caller's process group at the time of the call.

idtype == **P_ALL**
Wait for any child; *id* is ignored.

The child state changes to wait for are specified by ORing one or more of the following flags in *options*:

WEXITED
Wait for children that have terminated.

WSTOPPED
Wait for children that have been stopped by delivery of a signal.

WCONTINUED
Wait for (previously stopped) children that have been resumed by delivery of **SIGCONT**.

The following flags may additionally be ORed in *options*:

WNOHANG
As for **waitpid**().

WNOWAIT
Leave the child in a waitable state; a later wait call can be used to again retrieve the child status information.

Upon successful return, **waitid**() fills in the following fields of the *siginfo_t* structure pointed to by *infop*:

si_pid The process ID of the child.

si_uid The real user ID of the child. (This field is not set on most other implementations.)

si_signo
Always set to **SIGCHLD**.

si_status
Either the exit status of the child, as given to **_exit**(2) (or **exit**(3)), or the signal that caused the child to terminate, stop, or continue. The *si_code* field can be used to determine how to interpret this field.

si_code Set to one of: **CLD_EXITED** (child called **_exit**(2)); **CLD_KILLED** (child killed by signal); **CLD_DUMPED** (child killed by signal, and dumped core); **CLD_STOPPED** (child stopped by signal); **CLD_TRAPPED** (traced child has trapped); or **CLD_CONTINUED** (child continued by **SIGCONT**).

If **WNOHANG** was specified in *options* and there were no children in a waitable state, then **waitid**() returns 0 immediately and the state of the *siginfo_t* structure pointed to by *infop* depends on the implementation. To (portably) distinguish this case from that where a child was in a waitable state, zero out the *si_pid* field before the call and check for a nonzero value in this field after the call returns.

POSIX.1-2008 Technical Corrigendum 1 (2013) adds the requirement that when **WNOHANG** is specified in *options* and there were no children in a waitable state, then **waitid**() should zero out the *si_pid* and *si_signo* fields of the structure. On Linux and other implementations that adhere to this requirement, it is not necessary to zero out the *si_pid* field before calling **waitid**(). However, not all implementations follow the POSIX.1 specification on this point.

RETURN VALUE

wait(): on success, returns the process ID of the terminated child; on failure, −1 is returned.

waitpid(): on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by *pid* exist, but have not yet changed state, then 0 is returned. On failure, −1 is returned.

waitid(): returns 0 on success or if **WNOHANG** was specified and no child(ren) specified by *id* has yet changed state; on failure, −1 is returned.

On failure, each of these calls sets *errno* to indicate the error.

ERRORS

EAGAIN
The PID file descriptor specified in *id* is nonblocking and the process that it refers to has not terminated.

ECHILD
(for **wait**()) The calling process does not have any unwaited-for children.

ECHILD
(for **waitpid**() or **waitid**()) The process specified by *pid* (**waitpid**()) or *idtype* and *id* (**waitid**()) does not exist or is not a child of the calling process. (This can happen for one’s own child if the action for **SIGCHLD** is set to **SIG_IGN**. See also the*Linux Notes* section about threads.)

EINTR
WNOHANG was not set and an unblocked signal or a **SIGCHLD** was caught; see **signal**(7).

EINVAL
The *options* argument was invalid.

ESRCH
(for **wait**() or **waitpid**()) *pid* is equal to **INT_MIN**.

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001.

NOTES

A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by **init**(1), (or by the nearest "subreaper" process as defined through the use of the **prctl**(2) **PR_SET_CHILD_SUBREAPER** operation); **init**(1) automatically performs a wait to remove the zombies.

POSIX.1-2001 specifies that if the disposition of **SIGCHLD** is set to **SIG_IGN** or the **SA_NOCLDWAIT** flag is set for **SIGCHLD** (see **sigaction**(2)), then children that terminate do not become zombies and a call to **wait**() or **waitpid**() will block until all children have terminated, and then fail with *errno* set to **ECHILD**. (The original POSIX standard left the behavior of setting **SIGCHLD** to **SIG_IGN** unspecified. Note that even though the default disposition of **SIGCHLD** is "ignore", explicitly setting the disposition to **SIG_IGN** results in different treatment of zombie process children.)

Linux 2.6 conforms to the POSIX requirements. However, Linux 2.4 (and earlier) does not: if a **wait**() or **waitpid**() call is made while **SIGCHLD** is being ignored, the call behaves just as though **SIGCHLD** were not being ignored, that is, the call blocks until the next child terminates and then returns the process ID and status of that child.

Linux notes

In the Linux kernel, a kernel-scheduled thread is not a distinct construct from a process. Instead, a thread is simply a process that is created using the Linux-unique **clone**(2) system call; other routines such as the portable **pthread_create**(3) call are implemented using **clone**(2). Before Linux 2.4, a thread was just a special case of a process, and as a consequence one thread could not wait on the children of another thread, even when the latter belongs to the same thread group. However, POSIX prescribes such functionality, and since Linux 2.4 a thread can, and by default will, wait on children of other threads in the same thread group.

The following Linux-specific *options* are for use with children created using **clone**(2); they can also, since Linux 4.7, be used with **waitid**():

__WCLONE
Wait for "clone" children only. If omitted, then wait for "non-clone" children only. (A "clone" child is one which delivers no signal, or a signal other than **SIGCHLD** to its parent upon termination.) This option is ignored if**__W ALL** is also specified.

__WALL (since Linux 2.4)
Wait for all children, regardless of type ("clone" or "non-clone").

__WNOTHREAD (since Linux 2.4)
Do not wait for children of other threads in the same thread group. This was the default before Linux 2.4.

Since Linux 4.7, the **__WALL** flag is automatically implied if the child is being ptraced.

C library/kernel differences

wait() is actually a library function that (in glibc) is implemented as a call to **wait4**(2).

On some architectures, there is no **waitpid**() system call; instead, this interface is implemented via a C library wrapper function that calls **wait4**(2).

The raw **waitid**() system call takes a fifth argument, of type *struct rusage* *. If this argument is non-NULL, then it is used to return resource usage information about the child, in the same manner as **wait4**(2). See **getrusage**(2) for details.

BUGS

According to POSIX.1-2008, an application calling **waitid**() must ensure that *infop* points to a *siginfo_t* structure (i.e., that it is a non-null pointer). On Linux, if *infop* is NULL, **waitid**() succeeds, and returns the process ID of the waited-for child. Applications should avoid relying on this inconsistent, nonstandard, and unnecessary feature.

EXAMPLES

The following program demonstrates the use of **fork**(2) and **waitpid**(). The program creates a child process. If no command-line argument is supplied to the program, then the child suspends its execution using **pause**(2), to allow the user to send signals to the child. Otherwise, if a command-line argument is supplied, then the child exits immediately, using the integer supplied on the command line as the exit status. The parent process executes a loop that monitors the child using **waitpid**(), and uses the W*() macros described above to analyze the wait status value.

The following shell session demonstrates the use of the program:

```
$ ./a.out &
Child PID is 32360
[1] 32359
$ kill −STOP 32360
stopped by signal 19
$ kill −CONT 32360
continued
$ kill −TERM 32360
killed by signal 15
[1]+  Done                    ./a.out
$
```

Program source

```
#include <sys/wait.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
    pid_t cpid, w;
    int wstatus;

    cpid = fork();
    if (cpid == −1) {
```

```
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {            /* Code executed by child */
        printf("Child PID is %jd\n", (intmax_t) getpid());
        if (argc == 1)
            pause();                    /* Wait for signals */
        _exit(atoi(argv[1]));

    } else {                    /* Code executed by parent */
        do {
            w = waitpid(cpid, &wstatus, WUNTRACED | WCONTINUED);
            if (w == −1) {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }

            if (WIFEXITED(wstatus)) {
                printf("exited, status=%d\n", WEXITSTATUS(wstatus));
            } else if (WIFSIGNALED(wstatus)) {
                printf("killed by signal %d\n", WTERMSIG(wstatus));
            } else if (WIFSTOPPED(wstatus)) {
                printf("stopped by signal %d\n", WSTOPSIG(wstatus));
            } else if (WIFCONTINUED(wstatus)) {
                printf("continued\n");
            }
        } while (!WIFEXITED(wstatus) && !WIFSIGNALED(wstatus));
        exit(EXIT_SUCCESS);
    }
}
```

SEE ALSO

_exit(2), **clone**(2), **fork**(2), **kill**(2), **ptrace**(2), **sigaction**(2), **signal**(2), **wait4**(2), **pthread_create**(3), **core**(5), **credentials**(7), **signal**(7)

COLOPHON

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.