Betriebssysteme und Systemsoftware *Andrés Montoya, 405409*
Übungsblatt 03 *Til Mohr, 405959*
Tutorium 1 *Marc Ludevid Wulf, 405401*

Aufgabe 3.1

a) Siehe prozesse.c.

b) fprintf nutzt im Gegensatz zu write einen Buffer. Dieser muss mittels fflush "losgeschickt" werden. Dies ist bei write nicht notwendig, da der übergebene String direkt dem Kernel übergeben wird um diesen zu printen. Bei mehreren Ausgaben ist fprintf besser in dem Sinn, dass es die String zusammenfasst und diese mit einem fflush alle zussammen ausgedruckt werden können. Da bei write jedes mal, das etwas gedruckt wird ein syscall ausgeführt wird ist es in diesem Fall langsamer. Soll eine Ausgabe so schnell wie möglich geprintet werden sollte man write nutzen. In jedem anderen Fall sollte fprintf und fflush effizienter sein.

c) Zombie-Prozesse sind Prozesse die zwar geendet haben, aber der zugehörige Parent-Prozess den Exit-Status noch nicht gelesen hat. Das heisst, dass noch nicht wait() aufgerufen wurde. Sobald der exit-Status gelesen wurde kann der Prozess dann von der Prozess-Tabelle entfernt werden.

d) Ein einfaches Beispiel könnte das Senden von Signalen zur Kommunikation mit dem Parent-Prozess sein, welches die PID benötigt.

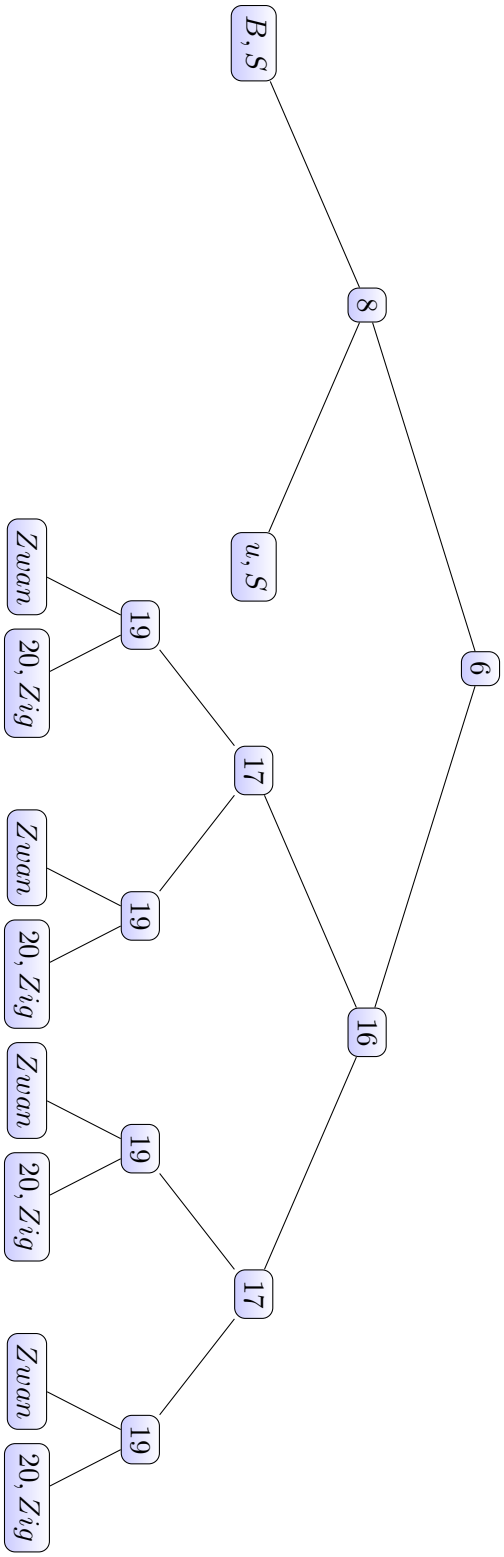
e) Siehe letter_count.c.

Aufgabe 3.2

a) In den Blättern stehen alle Ausgaben, die von dem jeweiligen Prozess erzeugt wurden. In den Knoten stehen die Zeilennummern, an denen die $fork$-Anweisung ausgeführt wurde.

Betriebssysteme und Systemsoftware
Übungsblatt 03
Tutorium 1

Andrés Montoya, 405409
Til Mohr, 405959
Marc Ludevid Wulf, 405401



b) u wird vor B ausgegeben, da der Prozess, der B ausgibt, ein Kindprozess von dem Prozess ist, der u ausgibt. In Zeile 8 wird dieser Prozess erstellt. Der Elternprozess gibt aber fast sofort später u aus, während der Kindprozess noch gestartet wird.

c) `wait(NULL)` warted auf die Terminierung *eines* Kindprozesses und gibt die PID des jeweiligen Kindprozesses zurück. Wenn `wait(NULL)` 0 zurückgibt, bedeutet dies, dass bereits alle Kindprozesse terminiert sind. Fügt man also zum Ende von `main` die Zeile `while(wait(NULL) > 0);` hinzu, so terminiert erst der Root-Prozess, wenn alle seiner Kinder terminiert sind. Dies gilt analog auch für alle Kindprozesse, die erst terminieren, wenn deren Kinder terminiert sind.

Aufgabe 3.3

a)

	Vorteile	Nachteile
Pipes	• Vereinfachte Synchronisation zwischen Prozessen	• Einseitiger Informationsfluss • Langsame Kommunikation • Prozesse können freezen
Shared Memory	• Beschleunigung der Kommunikation • Beidseitige Informationsfluss möglich • Prozesse unabhängig von einander → freezen nicht	• Erfordert Synchronisation zwischen Prozessen

b) Named Pipes haben die gleichen Vorteile und Nachteile wie normale Pipes, das heisst, dass eine Synchronisation zwischen den Prozessen nicht notwendig ist, andererseits ist die Kommmunikation nur einseitig und es kann zum Freezen eines Prozesses kommen. Named Pipes haben einen Eintrag im Dateisystem, also eine Kennung und sind durch Zugriffrechte identifizierbar.

Während Pipes Bytestreams benutzen, verwenden Messages eine vordefinierte Länge und können getyped sein. Message Passing ist ausserdem geeigent für Kommunikation zwischen mehreren Prozessen.

Shared Memory nutzt gemeinsame Speicherbereiche die geschützt werden müssen. Der Zugriff muss ausserdem synchronisiert werden. Dafür sind sie jedoch effizienter und beschleunigen somit die Kommmunikation.

c) Ein Prozess bezeichnet ein Programm im Stadium der Ausführung. Dazu wird jedem Prozess ein Process Control Block (PCB) zugewiesen, der aktuelle Werte und Zustände des Prozesses beinhaltet. Hierzu zählen beispielsweise alle Register, die Daten auf dem Arbeitsspeicher und vom Prozess geöffnete Dateien. Zudem besitzt jede PCB einen Programmzähler, wodurch erst die simultane Abarbeitung von Prozessen auf einer CPU ermöglicht wird.
Ein Thread ist eine Erweiterung des Prozess-Konzepts. Ein Prozess kann mehrere Threads erstellen, was eine Granulierung in der Strukturierung ermöglicht. Da jeder Thread einen eigenen Kontrollfluss besitzt, kann man hierdurch unabhängige Teilaufgaben gleichzeitig bearbeiten. Zudem teilen sich die Threads eines Programmes einige Werte und Zustände, neben der CPU-Zeit. All dies kann zu einer effizienteren Abarbeitung mit Threads als ohne führen.

d) Zu viele Threads sind schlecht, da diese System resources verbrauchen, wie zum Beispiel CPU-time oder Kernel-Funktionsaufrufen, wodurch auch Bugs enstehen können. Zudem fügen sie Komplexität zum Code hinzu, was Verständlichkeit und Robusheit verschlechtert und dadurch das Debuggen erschweren.