BuS-Übungsgruppe 7 Übungsblatt 3 Felix Kiunke, 357322
Philipp Hochmann, 356148
Daniel Schleiz, 356092

3.1

a)
Siehe prozesse.c

b)
Bei write() handelt es sich um einen syscall, welcher ungepuffert (theoretisch) direkt in die Standardausgabe geschrieben wird. fprintf() ist jedoch gepufferter I/O, es sammelt sich also zuerst ein chunk an Daten, welcher dann schließlich geschrieben wird. In dem gegebenen Beispielprogramm sollen einzelne chars in kurzer Abfolge geprintet werden, weshalb fflush benötigt wird. Diese Funktion leert den Puffer manuell, wodurch der Pufferinhalt geschrieben wird, im gegebenen Programm also immer ein einzelner char. Bei write() ist ein "flushen" nicht notwendig, da im Prinzip mittels syscall sofort (bzw. so schnell wie möglich) geschrieben wird.

Im Allgemeinen sollte man eher fprintf() und, falls notwendig, fflush verwenden, da gepufferter I/O meistens schlichtweg effizienter ist, weil zum Einen die Daten effizienter übertragen werde, und zum anderen, je nach Anwendungsfall, write() mehr Syscalls verursacht, wenn zum Beispiel write innerhalb einer Loop aufgerufen wird, was wiederum mehr Kontextwechsel erzeugt und somit zulasten der Performance ist, wohingegen sich bei fprintf unter Umständen die Daten zunächst im Puffer sammeln können, bis sie letzte Endes geschrieben werden, was weniger syscalls verursacht.

c)
Ein Zombie-Prozess ist ein Prozess, welcher beendet ist, aber noch in der Prozesstabelle steht. Dies kommt zum Beispiel zu Stande, indem ein Kindprozess vor seinem Vaterprozess fertig ist, wodurch der Kindprozess zwar beendet ist, aber weiter in der Prozesstabelle steht und dabei geringfügig Systemressourcen belegt, bis der Vaterprozess mittels z.B. wait() das Kind zum Löschen aus der Prozesstabelle freigibt.
Der "Zombie Zustand" eines Prozesses ist notwendig, da ein Vaterprozess sich so über die Art des Beendens des Kindes erkundigen kann, zum Beispiel erfolgreiches Ausführen, mit Fehlern oder Abstürzen usw. Diese Abfrage ist nur möglich, wenn der Kindprozess weiter als Zombie existiert.

d)
Dies wird so gemacht, da der Kernel bei Beendigung eines Kindprozesses ein besonderes Signal an den Vaterprozess schickt, um ihm dies mitzuteilen. Ohne die Information, was überhaupt der Vaterprozess vom entsprechenden Kindprozess ist, wäre dies nur schwer möglich. Zudem kann die Information über die PID des Vaters für die Interprozesskommunikation von Nutzen sein

falls es zum Beispiel aufgrund eines Anwendungsfalls notwendig ist, diese vom Kind aus einzurichten.

e)

Bitte 3.1e.c mit der Option -pthread compilieren, da pthreads verwendet werden! Leitet man den Output von prozesse.c auf 3.1e.c mittels pipe um, so wird der gewünschte Output auf der Konsole ausgegeben.
Betrachtet man nun z.B. mittels 'top' die zugewiesene Prozessorzeit der einzelnen Prozesse des Buchstabengenerators, so fällt auf, dass diese ungefähr gleich ist.

3.2

a) OUTPUT DES PROGRAMMS:

```
>>> Fuege neuen Studenten in die Liste ein: Eddard Stark [1234] ...
>>> Fuege neuen Studenten in die Liste ein: Jon Snow [5678] ...
>>> Fuege neuen Studenten in die Liste ein: Tyrion Lannister [9999] ...
>>> Test, ob die Matrikelnummer 1289 bereits erfasst wurde ...
    Matrikelnummer 1289 ist unbekannt
>>> Fuege neuen Studenten in die Liste ein: Daenerys Targaryen [1289] ...
>>> Loesche die Matrikelnummer 1234 ...
    Matrikelnummer 1234 geloescht
>>> Test ob die Studentenliste leer ist ...
    Die Studentenliste ist nicht leer
>>> Test, ob die Matrikelnummer 5678 bereits erfasst wurde ...
    Matrikelnummer 5678: Jon Snow
>>> Loesche die Matrikelnummer 9998 ...
    Matrikelnummer 9998 war nicht erfasst
>>> Loesche die Matrikelnummer 5678 ...
    Matrikelnummer 5678 geloescht
>>> Gebe alle erfassten Studenten aus ...
    Matrikelnummer 1289: Daenerys Targaryen
    Matrikelnummer 9999: Tyrion Lannister
```

b)
Siehe listen.c

3.3

a)
Sowohl Pipes als auch Shared Memory werden zur IPC genutzt.
Ein großer Vorteil von Pipes ist die relativ leichte Verwaltung, da nach dem Anlegen der Pipe der Kernel wichtige Verwaltungsaufgaben übernimmt, wie z.B. die Synchronisation , wodurch

mehrere Prozesse gleichzeitig auf die Pipe zugreifen können.
Außerdem gibt es sogenannte Named Pipes, welche das eigentliche Konzept der Pipes erweitern und keinem Prozess fest zugeordnet sind und somit in ihrer Nutzung flexibler sind, da sie sich im Prinzip wie eine Datei verhalten.
Pipes bringen jedoch auch Nachteile mit sich: Zunächst sind (zumindest reguläre) Pipes im Wesentlichen nur auf zwei Kommunikationspartner ausgerichtet. Zwar können an einer Pipe durch z.B. fork() Aufrufe mehr als nur zwei Prozesse hängen, jedoch sind Pipes für solche Fälle nicht wirklich gemacht, da z.B. etwas einmal aus der Pipe gelesenes danach aus der Pipe verschwindet, was unter Umständen nicht gewollt ist.
Vor allem aber finden bei Pipes immer Kontextwechsel in den Kernel-Space statt, da diese eben Kernel verwaltet sind. Dies verbraucht wichtige Systemressourcen, weshalb Pipes nicht unbedingt sehr schnell/effizient sind.

Bei Shared Memory kann ein bestimmter Speicherbereich von mehreren Prozessen gleichzeitig genutzt werden zur IPC, indem dieser Speicherbereich in den Speicher der jeweiligen Prozesse gemapt wird. Daraus folgt, dass Shared Memory einen asynchronen Zugriff auf gemeinsame Daten zwischen Prozessen ermöglichen welcher insbesondere schnellere IPC als Pipes ermöglicht.
Zudem, wie bereits impliziert, ist Shared Memory vor allem für mehrere Prozesse im Vergleich zu Pipes geeignet, da ein Prozess lediglich genug Speicher braucht, um dann den Shared Memory in seinen eigenen mapen zu lassen.
Die Vorteile bringen aber auch Nachteile mit sich: Zwar können sich viele Prozesse am Shared Memory beteiligen, jedoch können bei nicht synchronisiertem Zugriff von Prozessen darauf ungewünschte Seiteneffekte eintreten.
Da der Zugriff also je nach Anwendung synchronisiert werden sollte, folgt daraus, dass Shared Memory einen eher höheren Verwaltungsaufwand haben als Pipes.

b)
Je nach Pipe-Art liegt eine Pipe mehr oder weniger als Datei im Dateisystem vor, jedoch ist letzten Endes eine Pipe vom Konzept eher ein Datenstrom während eine Datei nunmal eine Datei ist.
Zum Einen ist eine Pipe eine temporäre Datei, nachdem also der Zugriff auf die Pipe beendet ist und die Pipe geschlossen wird, wird die Datei gelöscht, während bei einer "normalen" Datei ein Ende des Zugriffs nicht das Löschen der Datei mit sich zieht.
Zum Anderen hat eine Pipe das FIFO-Prinzip, das heißt, dass Sachen, welche zuerst reingeschrieben wurden, auch zuerst gelesen werden, wie bei einer normalen Datei. Jedoch verschwinden gelesene Daten nach dem Lesezugriff aus der Pipe, da FIFO dies so beschreibt, was offensichtlich bei einer normalen Datei nicht der Fall ist.

c)
Im Wesentlichen besitzt ein Prozess einen eigenen Speicherbereich, eine gewisse zugeteilte CPU-Zeit je nach Anwendung, sonstige Ressourcen und ermöglicht zunächst einmal einen einzigen Kontrollfluss.
Ein Thread ist ein sogenannter "Lightweight Process" existiert innerhalb eines Prozesses. Mehrere Threads ermöglichen mehrere Kontrollflüsse innerhalb eines Prozesses. Während ein Prozess einen eigenen Speicherbereich, teilen sich Threads zumindest den Addressbereich eines Prozesses und auch sonst einige Speicherbereiche des Prozesses, aber haben zum Beispiel einen eigenen Stack, zudem müssen sich Threads die CPU-Zeit des zugrunde liegenden Prozesses teilen. Da sich zum Beispiel Threads globale Variablen teilen gestaltet sich die Kommunikation von Threads (innerhalb eines Prozesses) meist einfacher als die Kommunikation zwischen Prozessen.

d)
Sowohl User-Threads als auch Kernel-Threads haben ihren eigenen Stack; User-Threads verstecken sich hinter einem Prozess und haben ihre eigenen Stacks innerhalb des Addressraums des Prozesses, lediglich der Kernel sieht diese Threads nicht, sondern nur einen single-threaded Prozess, dies beeinflusst jedoch nicht den Speicher der Threads.
Kernel-Threads haben ebenfalls sicherlich eigene Stacks, da beim Wechseln in einen Kernel-Thread ein Kontextwechsel notwendig ist, was impliziert, dass ein Kernel-Thread einen eigenen Stack besitzt.