Gruppe **20**
Betriebssysteme und Systemsoftware

Patrick Arens, 377536
Georg Manthey 376854
Benedikt Gerlach, 376944

Übung 3

Aufgabe 3.1

a)

Bei einer Pipe können Daten immer nur in eine Richtung fließen. Ein Prozess kann praktisch immer nur in eine Pipe schreiben oder aus einer Pipe lesen. Um gleichzeitig lesen und schreiben zu können werden mindestens 2 Pipes benötigt. Bei großen Prozessanzahlen mit hoher Kommunikation werden demnach exponentiell viele Pipes benötigt –> mehr Aufwand, komplexere Strukturen.

(Pipes: Kann zwischen Blockierend und nicht Blockierend wechseln, muss nicht wieder frei gegeben werden, automatisch an Kinder vererbbar)

Bei einem Shared Memory wird ein vom Kernel verwalteter Speicherbereich verwendet, auf den mehrere Prozesse zusammen Zugriff haben. Hierbei ist zu beachten, dass damit keine Kollision oder anderer Fehler beim Schreiben oder lesen von einem Prozess auf dem Shared Memory auftritt, alle Prozesses die auf diesen Speicher Zugriff haben synchronisiert werden müssen. Diese Synchronisation ist zwar aufwändiger als einzelne Pipes zu benutzen, dafür aber konstant bei auch sehr großen Prozessanzahlen.

(Shared memory: Kann structures speichern, kann nicht blockiert werden -> kann von so vielen Programmen genutzt werden wie man will)

Daraus folgt insgesamt ist für größere Prozesskommunikation ein Shared Memory sinnvoller und für kleinere Kommunikation zwischen wenigen Prozessen ist es einfacher bzw. schneller Pipes zu verwenden.

b)

Der wesentliche Unterschied zwischen Dateien und benannten Pipes ist, dass benannte Pipes immer nur eine Schreiben/ Lesen Operation ausühren bevor sie sich wieder schließen. Zudem können Daten die einmal hineingeschrieben wurden auch nur einmal von einem anderen Prozess ausgelesen werden. Zuletzt können bei bennanten Pipes auch nur wie bei normalen Pipes nur ein Prozess gleichzeitig auf diesem arbeiten. Bei einer Datei könnten es mehrere Gleichzeitig an unterschiedlichen Stellen.

c) Prozesse: laufen in shared memory, Kann kein Teil eines Threads sein, kann ein oder mehr Threads enthalten
Threads: laufen in seperaten Speicher Bereichen, Teil eines Prozesses

d) Werden User-Threads verwendet, gibt es nur einen Stack für den gesamten Prozess, da die Implementierung des Threads bereits im Programm passiert und der Kernel nur den gesamten Prozess sieht. Allerdings ist der Stack den der gesamte Prozess hat irgendwie eingeteilt, da jeder Thread einzeln abläuft. Diese Aufteilung wird aber vorher im Programm unternommen und der Kernel bekommt davon nichts mit.
Bei Kernel-Threads dagegen hat jeder Thread einen eigenen Stack, da hier die Threads durch den Kernel erstellt wurden. Hier kann auch durch den Kernel zwischen den Threads gewechselt werden (z.B. wenn ein Thread blockiert), das ist beim User-Thread nicht möglich.

Aufgabe 3.2

a)

In diesem Programm wird eine Kommunikation über ein Shared Memory ohne Prozesssynchronisation genutzt.

Zeile 18: Hier wird der Prozess mit der ID id an ein freies Shared Memory angehängt auf dem er arbeiten soll Im schritt danach wird dieses Shared Memory auf den Wert 0 initialsiert.

In der folgenden Schleife wird für eine vordefinierte Anzahl von „Kindern“ Kindsprozesse vom Vater gebildet. Für jeden dieser Kindsprozesse wird dann überprüft ob dieser erzeugt werden konnte. Wenn nein terminiert hier das Programm.

Nach dem ein Kindprozess erzeugt wurde, zählt er die gemeinsame Zählvariable und eine eigene Zählvariable solange hoch, bis der Wert `MAXCOUNT` erreicht wird. Danach gibt der jeweilige Kindsprozess noch den Wert seiner eigenen Zählvariable aus, also seinen Anteil am hochzählen der gemeinsamen Variable. Der Vaterprozess dagegen erzeugt weitere Kinder, da bei ihm der Wert von `pid[i]` nicht `0` ist.

Der Vaterprozess wartet dann auf das Terminieren all seiner Kindsprozesse.

Danach wird erst der Shared Memory vom Vaterprozess gelöst und im anschluss selbst gelöscht.

b)

1. Auffälligkeit: Die Summe der einzelnen Kindzprozesse ist ungleich der Anzahl von `MAXCOUNT`. Das liegt daran dass wenn ein Kindsprozess den Wert liest und dieser kleiner ist und dann darauf folgend von einem anderen Kindsprozess verändert wird, setzt der erste Kindsprozessbeim schreiben alles in der zwischenzeit geschehene zurück, was dazu führt das in der Summe mehr aller Prozesse ein größerer Wert als `MAXCOUNT` heraus kommt.

2. Auffälligkeit: Das Verhältnis zwischen den Anzahlen der einzelnen Prozesse ist in jedem Durchlauf komplett anders. Das heißt die Prozessverteilung und die Interrupts die auf diese Einwirken sind bei jedem Durchlauf andere und somit kommt es zu stetigen nicht vorraussagenden Verhältnissen zwischen den Anzahlen der Prozesse.