
Technische Informatik III:
Betriebssysteme und Rechnernetze WS 2007/08
Musterlösung zum Übungsblatt Nr. 2

Aufgabe 1: Begriffe

4 Punkte

Beschreiben Sie jeden der folgenden Begriffe durch maximal zwei Sätze: Prozess, Trace, Adressraum, Memory Management, Buddy System, Paging, Translation Lookaside Buffer, virtueller Speicher.

- Trace: Liste der Befehle, die für einen Prozess ausgeführt werden. Am Trace lässt sich das Verhalten des Prozesses erkennen (z.B. wann welcher Befehl ausgeführt wird).
- Paging ist eine Methode der Speicherverwaltung. Hier wird der virtuelle Speicher in kleinere, gleich große Teile unterteilt, die vom Betriebssystem bei Bedarf in die Seitenrahmen des Arbeitsspeicher geladen werden.
- Translation Lookaside Buffer: Ein Cache in der MMU, der häufig verwendete Address-Mappings speichert, um Zugriff auf die Page-Tabelle einzusparen.
- Virtueller Speicher: Ein Konzept, bei dem physikalische Adressen auf virtuelle Adressen umgesetzt werden. Damit ist es möglich, mehr Speicher zu adressieren, als tatsächlich im Arbeitsspeicher zu Verfügung steht - nicht verwendete Speicherblöcke werden vom Betriebssystem transparent auf den Sekundärspeicher ausgelagert. Virtueller Speicher: Konzept zur effizienteren Speicherverwaltung, bei der jeder Prozess sich so verhalten kann, als ob er den ganzen Speicher für sich alleine hat.

Aufgabe 2: Process Control Block

6 Punkte

Erläutern sie in wenigen Sätzen die drei Bereiche in die man den Inhalt des Process Control Blocks einteilen kann.

- Prozessvariablen:
Prozess ID, ID des Elternprozess, ID's der Kinder und User ID.
- Zustandsinformationen:
Registerinhalte und Stackpointer.
- Kontrollinformationen:
Informationen zum Speicher Management, Privilegien, Rechte an Betriebsmitteln, Interprozesskommunikation und Statusinformationen.

Aufgabe 3: Auslagerung

6 Punkte

Diskutieren Sie stichpunktartig die Vor- und Nachteile dreier Ersetzungsstrategien von Pages beim Swapping.

- First-In, first-out (FIFO)
Dieser Algorithmus ist einfach zu implementieren und hat kaum Overhead. Beachtet allerdings nicht das Prozessverhalten, so dass er in sich in der Praxis eine schlechte Performance aufweist.
- Clock oder second chance
Modifizierung von FIFO, benötigt extra Bit (R-Bit).
- Least recently used (LRU)
Nähert das Optimum durch das Lokalitätsprinzip an. Hoher Aufwand.

Aufgabe 4: Fifty/Fifty

8 Punkte

Entscheiden Sie, ob folgenden Aussagen zutreffend oder nicht zutreffend sind, und begründen Sie Ihre Entscheidung:

- **Prozesswechsel sind nach Möglichkeit zu vermeiden, da sie interaktive Benutzeranfragen verlangsamen.**
Falsch. Ein Prozess, der auf Benutzeranfragen wartet, tut nichts, da können andere Prozesse die Rechenzeit besser nutzen. Ohne Prozesswechsel gibt es keine Fairness der Rechenzeitaufteilung unter den Prozessen.
- **Die Anzahl von Kindern, die ein Prozess haben kann, ist vom Betriebssystem vorgegeben.**
Richtig. Die Anzahl ist durch die Implementierung der Verwaltungsstrukturen oder des Typs des PIDs im Betriebssystem gegeben.
- **Gemeinsamer Zugriff von mehreren Prozessen auf denselben Speicherbereich ist aus Sicherheitsgründen zu vermeiden.**
Falsch. Zwischen mehreren Prozessen geteilter Speicher ist einer der grundlegenden Mechanismen der Inter-Prozess Kommunikation.
- **Ausgelagerte Speichersegmente müssen nur bei Schreibzugriffen in den Primärspeicher geladen werden.**
Falsch. Auch zum Lesen müssen sie wieder eingelagert werden.

Aufgabe 5: C Syntax

8 Punkte

Kommentieren Sie das folgende Programm, ein Kommentar pro Zeile. Kommentieren Sie ebenfalls vor jeder Funktion deren Funktion.

aufgabe2.5.c

```
1  #include <stdio.h> /* Standardbibliothek für Ein-/Ausgabe*/
2  #include <stdlib.h> /* Standardbibliothek enthält malloc() und free() */
3
4  /** @struct ns
5   * @brief Struktur für ein Listenelement die eine ganze Zahl als Datum enthält
6   */
7  typedef struct ns {
8      int data; /* Datum */
9      struct ns *next; /* Zeiger zum nächsten Element */
10 } node; /* typedef für neuen Typ node */
11
12 /** @fn node *list_add(node **head, int i)
13 * Fügt das gegebene Datum in vorne die gegebene Liste ein.
14 *
15 * @param head Kopf der Liste
16 * @param i Einzufügendes Datum
17 * @return Gibt neues Element bzw. Kopf der Liste zurück
18 */
19 node *list_add(node **head, int i) {
20     node *n = malloc(sizeof(node)); /* Speicherreservierung für das neue Element */
21     n->data = i; /* Speicherung des Datums im neuem Element */
22     n->next = *head; /* Setzte erstes Element der Liste als Nachfolgeelement des neuen */
23     *head = n; /* Setzte neues Element als neuen Kopf der Liste */
24     return n; /* Gebe neues Element, also Kopf der Liste zurück */
25 }
26
27 /** @fn void list_remove(node **head)
28 * Löscht das erste Element der übergebenen Liste
29 * und gibt dessen Speicher frei.
30 *
31 * @param head Kopf der Liste
32 */
33 void list_remove(node **head) {
34     if(*head != NULL) { /* Überprüfung, ob übergebenes Element existiert */
35         node *tmp = *head; /* Speichere Adresse des übergebenes Element */
36         *head = (*head)->next; /* Setze Kopf der Liste auf das nächste Element */
37     }
```

```

37     free(tmp);          /* Freigabe des Speichers */
38 }
39 }
40
41 /** @fn void list_print(node *n)
42  * Gibt die übergebene verkettete Liste auf die Kommandozeile aus.
43  *
44  * @param n Kopf der Liste
45  */
46 void list_print(node *n) {
47     while (n != NULL) { /* while-Schleife zur Iteration über die Liste */
48         /* Ausgabe des aktuellen Listenelements: Datum, Adresse des folgenden Elementes, und
49            Adresse des Elementes */
49         printf("[data: %d, next: %p] @ %p, ", n->data, n->next, n);
50         n = n->next; /* Gehe zum nächsten Listenelement */
51     }
52     printf("[NULL]\n"); /* Gebe Ende der Liste aus */
53 }
54
55 /** @fn int main(int argc, char *argv[])
56  * Einstiegsfunktion für den Compiler, zur Illustration
57  * der implementierten Funktionen zur verketteten Liste.
58  *
59  * @param argc Anzahl der Argumente
60  * @param argv Feld für die Argumente, werden hier ignoriert
61  * @return Status der Terminierung
62  */
63 int main(int argc, char *argv[]) {
64     node *n = NULL; /* Deklariere neues Listenelemente bzw neue Liste */
65     list_print(n); /* Gebe Liste aus */
66     list_add(&n, 1); /* Füge "1" in die Liste ein */
67     list_add(&n, 2); /* Füge "2" in die Liste ein */
68     list_add(&n, 3); /* Füge "3" in die Liste ein */
69     list_print(n); /* Gebe Liste aus */
70     list_remove(&n); /* Lösche Kopf der Liste */
71     list_remove(&n); /* Lösche Kopf der Liste */
72     list_remove(&n); /* Lösche Kopf der Liste */
73     list_print(n); /* Gebe Liste aus */
74
75     return EXIT_SUCCESS; /* Austriegspunkt des Funnktion */
76 }

```

Aufgabe 6: Pointer

8 Punkte

Implementieren Sie in C folgende Operationen auf einfach verketteten Listen:

Testlauf:

```

nawab:~/ti3> ./aufgabe2.6
[NULL]
[data: 3, next: 0x804a018] @ 0x804a028,
[data: 2, next: 0x804a008] @ 0x804a018,
[data: 1, next: (nil)] @ 0x804a008,
[NULL]
[NULL]

```

Testlauf Aufgabe 6

```

Füge 9 hinzu: ok
Füge 2 hinzu: ok
Füge 3 hinzu: ok
Füge 7 hinzu: ok
Füge 4 hinzu: ok
Füge 100 hinzu: ok
Füge 10 hinzu: ok
[data: 10, next: 0x804a058] @ 0x804a068,
[data: 100, next: 0x804a048] @ 0x804a058,
[data: 4, next: 0x804a038] @ 0x804a048,
[data: 7, next: 0x804a028] @ 0x804a038,
[data: 3, next: 0x804a018] @ 0x804a028,

```

```
[data: 2, next: 0x804a008] @ 0x804a018,  
[data: 9, next: (nil)] @ 0x804a008,  
[NULL]
```

Test list_get

Hole 1. Element: ok

Hole 6. Element: ok

Hole 18. Element: ok

Test list_delete

```
[NULL]
```

Test list_add_at

Füge 17 hinzu: ok

Füge 8 hinzu: ok

Füge 45 hinzu: ok

```
[data: 17, next: 0x804a018] @ 0x804a008,  
[data: 8, next: 0x804a028] @ 0x804a018,  
[data: 45, next: (nil)] @ 0x804a028,  
[NULL]
```

Test list_remove_at

Lösche 9. Element: ok

```
[data: 17, next: 0x804a018] @ 0x804a008,  
[data: 8, next: 0x804a028] @ 0x804a018,  
[data: 45, next: (nil)] @ 0x804a028,  
[NULL]
```

Lösche 2. Element: ok

```
[data: 17, next: 0x804a028] @ 0x804a008,  
[data: 45, next: (nil)] @ 0x804a028,  
[NULL]
```

aufgabe2.6.c

```
1  /**  
2  * @file aufgabe2.6.c  
3  *  
4  * TI III: Betriebssysteme und Rechnernetze  
5  * WS 2007/08  
6  * Übungsblatt Nr. 2,  
7  * Aufgabe6: Pointer  
8  *  
9  * Dieses Programm implementiert die geforderten Funktionen von Aufgabe 6 zu der  
10 * verketteten Liste.  
11 *  
12 * @author Miao Wang  
13 * @see http://cst.mi.fu-berlin.de/teaching/WS0708/19513-V-TI-III/index.html  
14 * @date 2007-10-21  
15 */  
16 #include <stdio.h> // Bibliothek für Ein- und Ausgabe laden  
17 #include <stdlib.h> // Standard Bibliothek laden  
18  
19 typedef struct ns { // Erzeugen eines struct Typs mit  
20     int data;        // einem int data  
21     struct ns *next; // und einem pointer auf das nächste Element  
22 } node;              // als einfach verkettete Liste  
23  
24 node *list_add(node **head, int i) { // Funktion zum anhängen eines Knotens vorne  
25     node *n = malloc(sizeof(node)); // Speicher reservieren für ein node  
26     n->data = i;                    // zuweisen des Datenwerts  
27     n->next = *head;                // zuweisen des Nachfolgerknotens  
28     *head = n;                      // erstes Element ist nun n  
29     return n;                       // Rückgabe des neuen Knotens  
30 }  
31  
32 void list_remove(node **head) { // Funktion zum löschen des ersten Knotens  
33     if(*head != NULL) {          // falls es eine Liste gibt  
34         node *tmp = *head;        // speicher head temporär ab
```

```

35     *head = (*head)->next; // zweites Element ist nun head
36     free(tmp);           // gebe Speicher vom tmp frei
37 }
38 }
39
40 void list_print(node *n) { // Funktion zur Ausgabe einer Liste ab n
41     while (n != NULL) { // solange nicht das Ende erreicht ist
42         printf("[data: %d, next: %p] @ %p, ", n->data, n->next, n); //Ausgabe
43         n = n->next; // gehe zum nächsten Knoten
44     }
45     printf("[NULL]\n"); // Ende anzeigen
46 }
47
48 // Aufgabe 6
49 node *list_get(node **head, int i) { // Funktion um ein Element zurückzugeben
50     if(*head == NULL) return NULL; // falls es keine Liste gibt, gibt es auch kein Element
51
52     int j; // Hilfszähler j
53     node *tmp = *head; // Lafelement tmp startet bei head
54
55     for(j=1; j<i; j++) { // durchläuft bis i-te Stelle
56         if(tmp->next == NULL) return NULL; // wenn Ende erreicht gebe NULL zurück
57         tmp = tmp->next; // sonst laufe weiter
58     }
59     return tmp; // i-te Stelle erreicht, gebe zurück
60 }
61
62 void list_delete(node **head) { // Funktion zum Löschen der gesamten Liste
63     while(*head != NULL) { // solange Liste nicht leer
64         list_remove(head); // bediene dich an der vordefinierten Methode
65     }
66 }
67
68 void list_add_at(node **head, node *n, int i) { // Funktion zum Hinzufügen eines Elements an i-
        ter Stelle
69     if(i <= 1 || *head == NULL) { // wenn i kleiner oder gleich als 1 oder Liste leer
70         n->next = *head; // benutze Funktion um an Kopf hinzuzufügen
71         *head = n;
72         return; // fertig
73     }
74
75     int j; // sonst erstelle Hilfszähler j
76     node *tmp = *head; // Lafelement tmp, startet bei head
77
78     for(j=2; j<i; j++) { // durchläuft bis i-te Stelle
79         if(tmp->next == NULL) { // wenn Ende erreicht springe raus
80             break;
81         }
82         tmp = tmp->next; // sonst laufe weiter
83     }
84
85     n->next = tmp->next; // i-te Stelle erreicht, füge Element ein
86     tmp->next = n;
87 }
88
89 node *list_remove_at(node **head, int i) { // Funktion zum Löschen eines Elements an i-ter
        Stelle
90     if(i < 1 || *head == NULL) return NULL; // ungültige Eingaben abfangen
91
92     if(i == 1) { // wenn i 1 ist
93         node *tmp = *head; // entferne Kopf
94         *head = (*head)->next;
95         return tmp;
96     }
97
98     int j; // sonst erstelle Hilfszähler j
99     node *tmp = *head; // Lafelement tmp, startet bei head
100
101     for(j=2; j<i; j++) { // durchläuft bis i-te Stelle
102         if((tmp->next)->next == NULL) return NULL; // wenn Ende erreicht ist nichts zu tun
103         tmp = tmp->next; // sonst laufe weiter
104     }
105
106     node *tmp2 = tmp->next; // i-te Stelle erreicht, biege Zeiger um
107     tmp->next = tmp2->next;
108     return tmp2; // und lösche Element
109 }
110
111 int main(int argc, char *argv[]) { // Hauptfunktion
112     node *n = NULL; // leere Liste erstellen
113     list_print(n); // Liste ausgeben
114     list_add(&n, 1); // 1 in Liste einfügen

```

```

115 list_add(&n, 2); // 2 in Liste einfügen
116 list_add(&n, 3); // 3 in Liste einfügen
117 list_print(n); // Liste ausgeben
118 list_remove(&n); // Kopfelement entfernen
119 list_remove(&n); // Kopfelement entfernen
120 list_remove(&n); // Kopfelement entfernen
121 list_print(n); // Liste ausgeben
122
123
124 //Testlauf Aufgabe 6
125 printf("\nTestlauf Aufgabe 6\n");
126 list_add(&n, 9);
127 printf("Füge 9 hinzu:\t%s\n", list_get(&n, 1)->data == 9 ? "ok" : "fehler");
128 list_add(&n, 2);
129 printf("Füge 2 hinzu:\t%s\n", list_get(&n, 1)->data == 2 ? "ok" : "fehler");
130 list_add(&n, 3);
131 printf("Füge 3 hinzu:\t%s\n", list_get(&n, 1)->data == 3 ? "ok" : "fehler");
132 list_add(&n, 7);
133 printf("Füge 7 hinzu:\t%s\n", list_get(&n, 1)->data == 7 ? "ok" : "fehler");
134 list_add(&n, 4);
135 printf("Füge 4 hinzu:\t%s\n", list_get(&n, 1)->data == 4 ? "ok" : "fehler");
136 list_add(&n, 100);
137 printf("Füge 100 hinzu:\t%s\n", list_get(&n, 1)->data == 100 ? "ok" : "fehler");
138 list_add(&n, 10);
139 printf("Füge 10 hinzu:\t%s\n", list_get(&n, 1)->data == 10 ? "ok" : "fehler");
140 list_print(n);
141
142 //Test list_get
143 printf("\nTest list_get\n");
144 node *a = list_get(&n, 1);
145 printf("Hole 1. Element:\t%s\n", a->data == 10 ? "ok" : "fehler");
146 node *b = list_get(&n, 6);
147 printf("Hole 6. Element:\t%s\n", b->data == 2 ? "ok" : "fehler");
148 node *c = list_get(&n, 18);
149 printf("Hole 18. Element:\t%s\n", c == NULL ? "ok" : "fehler");
150
151 //Test list_delete
152 printf("\nTest list_delete\n");
153 list_delete(&n);
154 list_print(n);
155
156 //Test list_add_at
157 printf("\nTest list_add_at\n");
158 node *neu = malloc(sizeof(node));
159 neu->data = 17;
160 list_add_at(&n, neu, 1);
161 printf("Füge 17 hinzu:\t%s\n", list_get(&n, 1)->data == 17 ? "ok" : "fehler");
162 node *neu2 = malloc(sizeof(node));
163 neu2->data = 8;
164 list_add_at(&n, neu2, 2);
165 printf("Füge 8 hinzu:\t%s\n", list_get(&n, 2)->data == 8 ? "ok" : "fehler");
166 node *neu3 = malloc(sizeof(node));
167 neu3->data = 45;
168 list_add_at(&n, neu3, 23);
169 printf("Füge 45 hinzu:\t%s\n", list_get(&n, 3)->data == 45 ? "ok" : "fehler");
170 list_print(n);
171
172 //Test list_remove_at
173 printf("\nTest list_remove_at\n");
174 node *x = list_remove_at(&n, 9);
175 printf("Lösche 9. Element:\t%s\n", x == NULL ? "ok" : "fehler");
176 list_print(n);
177 x = list_remove_at(&n, 2);
178 printf("\nLösche 2. Element:\t%s\n", x->data == 8 ? "ok" : "fehler");
179 list_print(n);
180
181 free(neu);
182 free(neu2);
183 free(neu3);
184
185 return EXIT_SUCCESS; // Ende
186 }

```

Aufgabe 7: Prozessabbild

4 Punkte

Betrachten Sie bei dem von Ihnen in Aufgabe 6 erstellten Programm die Speicheradressen der globalen, lokalen und dynamisch allozierten Variablen, sowie der Funktionen. Tragen Sie Die Adressen in ein Diagramm wie das folgende ein, und weisen Sie den den Bereichen Bezeichnungen wie Code-, Heap- und Stackspeicher zu.

Funktion (Code):	
list_add	0x80483f4
list_print	0x8048459
list_remove:	0x8048429
list_get:	0x80484a7
list_delete:	0x804852a
list_add_at:	0x804854a
list_remove_at:	0x80485be
main:	0x804869e
Elemente (Heap):	
1. Element:	0x804a048
2. Element:	0x804a038
3. Element:	0x804a028
4. Element:	0x804a018
5. Element:	0x804a008
Lokale Variablen (Stack):	
In main():	
node *n:	0xbfebe9f4
node *tmp:	0xbfebe9f0
int i:	0xbfebe9ec
In list_get():	
int j:	0xbfebe9d4
node *tmp:	0xbfebe9d0

Aufgabe 8: Prozesserzeugung

6 Punkte

Implementieren Sie in C ein Programm, dass mit dem System Aufruf *fork()* ein Kind-Prozesse erstellt. Dann soll der Kind-Prozess wiederum einen neuen Prozess erstellen und so weiter, bis eine Verschachtelungstiefe von zehn Prozessen erreicht ist. Danach sollen sich alle Prozesse wiederum in umgekehrter Reihenfolge beenden. Dabei soll sich jeder Prozess mit einer Nachricht „Prozess der Tiefe *x* wird beendet“ (*x* von 0 bis 9) verabschieden. Um dies zu erreichen, informieren Sie sich in den Manpages über die *wait*-Funktion.

Testlauf:

```
nawab:~/ti3> ./aufgabe2.8
Prozess der Tiefe 10 wird beendet
Prozess der Tiefe 9 wird beendet
Prozess der Tiefe 8 wird beendet
Prozess der Tiefe 7 wird beendet
Prozess der Tiefe 6 wird beendet
Prozess der Tiefe 5 wird beendet
Prozess der Tiefe 4 wird beendet
Prozess der Tiefe 3 wird beendet
Prozess der Tiefe 2 wird beendet
Prozess der Tiefe 1 wird beendet
Prozess der Tiefe 0 wird beendet
```

aufgabe2.8.c

```
1  /**
2  * @file aufgabe2.8.c
3  *
4  * TI III: Betriebssysteme und Rechnernetze
5  * WS 2007/08
6  * Übungsblatt Nr. 2,
7  * Aufgabe 8: Prozesserzeugung
8  *
9  * Dieses Programm erzeugt 10 Kindprozesse.
```

```

10  *
11  * @author Christian Grümme
12  * @see http://cst.mi.fu-berlin.de/teaching/WS0708/19513-V-TI-III/index.html
13  * @date 2007-10-21
14  */
15  #include <stdio.h>
16  #include <stdlib.h>
17  #include <unistd.h>
18  #include <sys/types.h> /* Für Typ pid_t */
19  #include <sys/wait.h> /* Für wait() */
20
21
22  /** @fn int main(int argc, char *argv[])
23   * Hauptfunktion erzeugt 1 Kindprozesse und wartet auf sie.
24   *
25   * @param argc Anzahl der Argumente
26   * @param argv Argumente, werden ignoriert
27   * @return Gibt bei Erfolg 0 aus.
28   */
29  int main(int argc, char **argv)
30  {
31      int status; /* Wert zum speichern des Returnstatus des Kindes */
32      int tiefe = 0; /* Aktuelle Tiefe */
33      pid_t pid = 0; /* Aktuelle Prozessid */
34
35      /* Kindprozess muss noch einen Kindprozess erzeugen */
36      while(pid == 0 && (tiefe < 9)) /* Schleife, für die Kindprozesse */
37      {
38          /* Erzeuge Kindprozess */
39          pid = fork();
40
41          /* Überprüfe, ob aktueller Prozess Kindprozess ist */
42          if(pid == 0)
43          {
44              /* Erhöhe die Tiefe */
45              ++tiefe;
46          }
47      }
48
49      /* Falls nicht Prozess der Tiefe 10, warte auf Kindprozess */
50      if(tiefe < 9)
51      {
52          wait(&status);
53      }
54
55      printf("Prozess der Tiefe %d wird beendet\n", tiefe);
56
57      return EXIT_SUCCESS;
58  }

```