

1. Begriffe

Trace Als Trace wird das Protokoll der Prozessausführung bezeichnet. D.h. zu jedem Prozess wird genau protokolliert, welcher Speicherbereich bzw. Befehl wann und wo aufgerufen wurde, so dass man sozusagen die Spur nachverfolgen kann.

Paging Als Paging wird eine Variante der Speicherverwaltung von Prozesse bezeichnet; hier wird der gesamte physikalische Speicher in gleich große "frames" unterteilt und jeder Prozess in viele logische "pages", die genau so groß wie ein Frame sind. Das OS verwaltet dann in Tabellen pro Prozess die Zuteilung von Page auf Frame (also von logischer auf physikalische Adresse).

Translation Lookaside Buffer In diesem Buffer, der zur MMU des Prozessors gehörig ist, werden die Page-Einträge der virtuellen Adressen verwaltet, die am häufigsten zuletzt angefragt wurden. D.h. hier wird zu den virtuellen Adressen tabellarisch die physikalische gespeichert. So kann das OS schnell nachprüfen, ob eine geforderte Adresse schon gespeichert ist (TLB hit) oder noch nicht (TLB miss) und gegebenenfalls nachladen.

Virtueller Speicher Als virtueller Speicher wird verwendeter Speicher bezeichnet, der außerhalb des Arbeitsspeichers liegt, so wie z.B. auf Festplatten. So kann dem Benutzer ein wesentlich größerer Arbeitsspeicher angeboten werden, als eigentlich vorhanden ist, was effektiveres Arbeiten, gerade mit mehreren Programmen ermöglicht.

2. Process Control Block

Man kann den PCB in die Bereiche "process identifier", "Process state information" und "Process control information" einteilen. Hierbei werden im Identifier Informationen gespeichert, die notwendig zum Identifizieren des Prozesses sind, wie eigene ID, aber auch ID von möglichen Eltern/Kindprozessen. In dem state information Bereich sind die momentanen Statuswerte eingetragen. Hier liegen die für den Anwender sichtbaren Register und die Control und Statusregister, zusätzlich ist der Stackpointer hier deponiert. Im Control-Information Bereich liegen dann Informationen, die über den Prozess selber hinausgehen wie Angaben zur Kommunikation mit verwandten Prozessen (IPC). Auch die Speicherverwaltung und die Privilegien des Prozesses wie Priorität oder Besitzerrechte sind hinterlegt.

3. Auslagerung

Die Least Recently Used Strategie schmeißt immer die in der vergangenen Zeit am wenigsten häufige benutzte Page raus. Dies erfordert also eine relativ hohe Intelligenz des Systems und ist somit nicht allzu einfach zu implementieren. Jedoch macht man sich hier die Lokalitätseigenschaften zu nütze, weswegen man in den meisten Fällen eine gute Hitrate bekommen wird.

Dagegen ist das FIFO-Prinzip simpler zu implementieren, da es sich um eine einfache Warteschlange handelt. Prozesse, die so am längsten drin sind, werden wieder rausgeschmissen.

Dies kann aber nachteilig sein, wenn man z.B. Schleifen hat, in denen hintereinander öfters verschiedene Speicherzugriffe erfolgen, und zwar so, dass nicht alle in die Warteschlange reinpassen. So wird dann in jeder Iteration unnötigerweise rausgeschmissen, obwohl man nach dem Lokalitätsprinzip vermuten könnte, dass wieder an die selbe Stelle referenziert wird. D.h. Trashing kann vermehrt auftreten.

Am schwierigsten zu implementieren ist die Optimal policy. Hier wird versucht, die Seite zu berechnen, die in Zukunft am längsten nicht gebraucht wird. Dies erfordert also ziemlich viel Logik und dementsprechend auch mehr Ressourcen. Zudem kann die Zukunft nicht exakt berechnet werden, auch wenn man durch die Lokalität einen guten Anhaltspunkt hat.

4. Fifty/Fifty

- *Prozesswechsel sind nach Möglichkeit zu vermeiden, da sie interaktive Benutzeranfragen verlangsamen.*
Falsch. Erst Prozesswechsel machen die Interaktivität möglich. Würde man nämlich auf Wechsel verzichten, dann könnte der Benutzer z.B. nicht die Mouse bedienen, während Musik gespielt wird. Man muss natürlich drauf achten, dass die Prozesswechsel nicht lange dauern, sonst wird mehr gewechselt als gerechnet, was der Interaktivität natürlich auch nicht zugute kommt.
- *Die Anzahl von Kindern, die ein Prozess haben kann, ist vom Betriebssystem vorgegeben.*
Ja. So existieren z.B. bei vielen UNIX Implementierungen im Kernel ein Array fest vorgegebener Größe von process Strukturen, mit dem die maximale Anzahl der vom System gleichzeitig verwaltbaren Prozesse eindeutig bestimmt ist.
- *Gemeinsamer Zugriff von mehreren Prozessen auf denselben Speicherbereich ist aus Sicherheitsgründen zu vermeiden.*
Jein. Einerseits könnte so ja ein Prozess dem anderen was in den Daten oder auch Programmcode ändern, und das will man ja nicht. Andererseits sollen Daten für IPC natürlich doch gesharet werden können.
- *Ausgelagerte Speichersegmente müssen nur bei Schreibzugriffen in den Primärspeicher geladen werden.*
Falsch. Bei Segmentation wird immer der ganze Prozess in den Primärspeicher geladen, wenn er aktiviert ist. Im Gegensatz dazu wird bei Paging immer nur die benötigte Seite nachgeladen, es ist aber auch hier egal, ob gelesen oder geschrieben werden soll. Sobald ausgelagert wurde, gibt es ein TLB-miss und es muss nachgeladen werden.

5. C Syntax

```
#include <stdio.h> // bindet die Bibliothek für Ein- und  
Ausgaben ein
```

```
#include <stdlib.h> // bindet die Standardbibliothek ein

typedef struct ns { // Definition eines Datentyps "ns"
    int data; // Unser Strukt enthält einen Int Wert data
    struct ns *next; // und einen Pointer auf ein anderes Struct
                    gleichen Typs mit Namen next
} node; // dieses ganze Konstrukt bekommt den Namen node

// Folgende Funktion fügt ein neues Element am Anfang der Liste
// ein und gibt den eingefügten Knoten zurück
node *list_add(node **head, int i) { // Funktion wird
    // deklariert mit Rückgabewert node. ein Pointer auf einen
    // Nodepointer und ein int werden als Parameter erwartet
    node *n = malloc(sizeof(node)); // legt einen pointer auf ein
    // Node-Objekt an, dem durch malloc Speicher in der von node
    // benötigten Größe reserviert
    n->data = i; // weist data von n den Wert i zu
    n->next = *head; // weist next von n die Adresse des Pointers
    // von head zu
    *head = n; // in den Pointer von head wird die Adresse von n
    // eingetragen
    return n; // gibt n zurück
}

// Funktion löscht das erste Element
void list_remove(node **head) { // Funktion wird deklariert,
    // sie erwartet wieder einen Pointer auf einen Pointer auf Node
    if(*head != NULL) { // prüft, ob es sich um eine leere Liste
        // handelt, dann verweist head nämlich noch nach NULL
        node *tmp = *head; // wenn nein, dann wird ein Pointer auf
        // ein Nodeobjekt angelegt, der die Adresse des head-Node
        // übergeben bekommt
        *head = (*head)->next; // der Pointer von head wird auf den
        // Nachfolger von head gesetzt
        free(tmp); // der Speicherplatz von tmp wird freigegeben
    }
}

// Funktion, die die ganze Liste ausgibt
void list_print(node *n) { // Deklaration der Funktion, sie
    // erwartet einen Pointer auf einen Knoten
    while (n != NULL) { // solange der Pointer nicht ins leere
        // zeigt
```

```
printf("[data:_%d, _next:_%p]_@_%p, _", n->data, n->next, n);  
    // gib den Inhalt als Integer aus und die Adresse vom  
    pointer des nächsten Elements. Zusätzlich wird die Adresse  
    des momentanen Knotens ausgegeben  
n = n->next; // setze den Pointer ein Element weiter, auf  
    den Nachfolger  
}  
printf("[NULL]\n"); // am Ende der Liste wird NULL ausgegeben  
}  
  
// Main-Methode, hier wird das Programm ausgeführt  
int main(int argc, char *argv[]) { // Main-Methode  
    node *n = NULL; // ein Pointer auf ein Nodeobjekt wird  
        erstellt, der auf NULL zeigt  
    list_print(n); // die Liste wird ausgegeben (NULL)  
    list_add(&n, 1); // es wird das Objekt 1 an n angehängen (da  
        ein Pointer auf Pointer erwartet wird, wird die Adresse  
        übergeben, nicht das Objekt selbst  
    list_add(&n, 2); // es wird Objekt 2 angehängen  
    list_add(&n, 3); // es wird Objekt 3 angehängen  
    list_print(n); // Liste wird wieder ausgegeben (enthält  
        3->2->1)  
    list_remove(&n); // es wird das 1. Element gelöscht, in dem  
        Fall die 3  
    list_remove(&n); // es wird das 1. Element gelöscht, jetzt die  
        2  
    list_remove(&n); // es wird wieder das 1. Element gelöscht,  
        jetzt 1  
    list_print(n); // Liste wird wieder ausgegeben (wieder NULL)  
    return EXIT_SUCCESS; //beendet das Programm  
}
```

6. Pointer

```
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct ns {  
    int data;  
    struct ns *next;  
} node;  
  
void list_add_at(node** head, node* n, int i) { // fügt einen  
    neues Element an der gegebenen Stelle hinzu oder ans Ende
```

```
    der Liste , falls die Liste nicht genug Elemente enthält. Das
    Element, das vorher an dieser Stelle stand (und alle
    folgenden) werden ein Platz nach hinten geschoben.
if(i == 1) { // Am Anfang einfügen
    node *tmp = *head;
    *head = n;
    n->next = tmp;
}
else { // irgendwo in der Liste einfügen
    int count = 1;
    node *n2 = *head;
    while(i != count+1) { //wann ist man eins vor gewünschter
        Position?
        n2 = n2->next;
        count++;
    }
    node *tmp = n2->next;
    n2->next = n;
    n->next = tmp;
}
}

node* list_remove_at(node** head, int i) {// löscht das Element
    an der gegebenen Stelle aus der Liste und gibt es zurück.
    Falls die Liste zu kurz ist, bleibt sie unverändert und die
    Funktion gibt NULL zurück.
    if(i == 1) // Am Anfang löschen
        list_remove(head);
    else { // mittendrin löschen
        int count = 1;
        node *n = *head;
        while(i != count+1) { //wann ist man eins vor der gewünschten
            Position?
            n = n->next;
            count++;
        }
        node *tmp = n->next;
        n->next = (n->next)->next;
        return tmp;
    }
}

void list_delete(node **head) { // löscht die gesamte Liste.
    while(*head != NULL) {
```

```
    node *tmp = *head;
    *head = (*head)->next;
    free(tmp);
}
}
```

```
node* list_get(node** head, int i) { // gibt das i-te Element
    der Liste zurück, und NULL, falls in der Liste zu wenig
    Elemente vorhanden sind
    int count = 1;
    node *n = *head;
    while(n != NULL) {
        if(i == count)
            return n;
        else {
            n = n->next;
            count++;
        }
    }
    return NULL;
}
```

8. Prozesserzeugung

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
    int i=0;
    int pid = fork(); // erster Kindprozess
    while(pid == 0 && i<9) { // solange die ID == 0, d.h. wir im
        Kindprozess sind und i < 9 ist
        i++; // erhöhe i
        if(i != 9) // wenn i nicht neun starte Kindprozess
            pid = fork();
    }
    if(pid != 0) { // wenn wir Elternprozess sind, dann wird
        gewartet, bis Kindprozess mit 0 beendet
        wait(0);
    }
    printf("Prozess der Tiefe %d wird beendet\n", i); // Prozess i
        beendet
    exit(0); // Prozess mit Code 0 beenden
}
```