altklausur-a1

Donnerstag, 5. Januar 2023 15:20

Aufgabe 1: Ankreuzfragen (30 Punkte)

1) Einfachauswahlfragen (22 Punkte)

Bei den Einfachauswahlfragen in dieser Aufgabe ist jeweils nur eine richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Antwort korrigieren, streichen Sie bitte die falsche Antwort mit drei waagrechten Strichen durch (⊗) und kreuzen die richtige an.

Lesen Sie die Frage genau, bevor Sie antworten.

a) Wie wird erkannt, dass eine Seite eines virtuellen Adressraums gerade ausgelagert ist? — 2 Punkte

- ○ Bei Programmen, die in virtuellen Adressräumen ausgeführt werden sollen, erzeugt der Compiler speziellen Code, der vor Betreten einer Seite die Anwesenheit überprüft und ggf. die Einlagerung veranlasst.
- ○ Im Seitendeskriptor wird ein spezielles Bit geführt, das der MMU zeigt, ob eine Seite eingelagert ist oder nicht. Falls die Seite nicht eingelagert ist, löst die MMU einen Trap aus.
- ○ Das Betriebssystem erkennt die ungültige Adresse vor Ausführung eines Maschinenbefehls und lagert die Seite zuerst ein bevor ein Fehler passiert.
- ○ Die MMU erkennt bei der Adressumsetzung, dass die physikalische Adresse ungültig ist und löst einen Trap aus.

不是发生在 adresseumsetzung 的时候，而是 发生在 program 执行的时候，发现 在 page table 里面的 presence bit 是 0 的时候，会 触发 demanding page。

Die virtuell adresse wird von MMU in die physikalische adresse umgesetzt, wenn eine seite ausgelagert , dann in der tabelle gibt es ein bit,

-----> page fault ----> demanding page, diese page wird noch mal eingelagert ----> tabelle bit wird geandert und program 第二次 被尝试 执行。

b) Welche der folgenden Aussagen über Schedulingverfahren ist richtig? — 2 Punkte

- ○ Bei preemptivem Scheduling sind Prozessumschaltungen unmöglich, wenn ein Prozess in einer Endlosschleife läuft.
- ○ Bei kooperativem Scheduling sind Prozessumschaltungen unmöglich wenn ein Prozess in einer Endlosschleife läuft, selbst wenn er bei jedem Schleifendurchlauf einen Systemaufruf macht.
- ○ Preemptives Scheduling ist für Mehrbenutzerbetrieb geeignet.
- ○ Kooperatives Scheduling ist für Steuerungssysteme mit Echtzeitanforderungen völlig ungeeignet.

the cpu will kick this process out

selbst entscheidet wann verlassen

jeder Benutzer hat einige "time slice"

有一些 比较 重要的 process, , 你就得等着 这个 process 完事，不能 interrupt

搞清 preemptive scheduling 和 kooperative scheduling 的 区别:

Process Dispatching: Time and Selection

- transitions to the ready state (READY) update the CPU ready list
  - a decision regarding the **placement** of the **process control block** is made
  - the result depends on the system's **CPU scheduling strategy**
- scheduling or rescheduling (dt. Einplanung/Umplanung) occurs, ...
  1. after a process has been created
  2. when a process yields control of the CPU
  3. if the event which is expected by a process has occurred
  4. as soon as an swapped-out process is resumed

A process can be **forced** to release the CPU → preemptive scheduling

- for example, by a timer interrupt ②

CPU entscheidet, process weg oder nicht
OS kann den Process entziehen

Cooperative scheduling **is a style of scheduling in which the OS never interrupts a running process to initiate a context switch from one process to another**. Processes must voluntarily yield control periodically or when logically blocked on a resource.

process entscheidet selbst, wann weg

bei preemptive Scheduling is es meistens mit "time slice " zusammen

time slice 用完之后，egal process fertig oder nicht， weg!

Time-slice Driven: Round Robin (RR)

- reduces the **penalization of processes with short CPU bursts** that occurs with FCFS:
  - the **processor time** is divided into **time slices** (dt. Zeitscheiben)
- when the time slice expires, a process change may take place
  - the interrupted process is pushed to the end of the ready list
  - the next process is taken from the ready list according to FCFS
- basis for CPU protection: timer interrupt enforces end of time slice
- the **time slice length** determines the effectiveness of this scheduling method
  - time slice
    - too long: degeneration of RR to FCFS
    - too short: high scheduling overhead
  - rule of thumb: slightly longer than the duration of a "typical interaction"

altklausur-a1

Donnerstag, 5. Januar 2023 15:20

Aufgabe 1: Ankreuzfragen (30 Punkte)

1) Einfachauswahlfragen (22 Punkte)

Bei den Einfachauswahlfragen in dieser Aufgabe ist jeweils nur eine richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Antwort korrigieren, streichen Sie bitte die falsche Antwort mit drei waagrechten Strichen durch (⊗) und kreuzen die richtige an.

Lesen Sie die Frage genau, bevor Sie antworten.

a) Wie wird erkannt, dass eine Seite eines virtuellen Adressraums gerade ausgelagert ist? — 2 Punkte

- ○ Bei Programmen, die in virtuellen Adressräumen ausgeführt werden sollen, erzeugt der Compiler speziellen Code, der vor Betreten einer Seite die Anwesenheit überprüft und ggf. die Einlagerung veranlasst.
- ○ Im Seitendeskriptor wird ein spezielles Bit geführt, das der MMU zeigt, ob eine Seite eingelagert ist oder nicht. Falls die Seite nicht eingelagert ist, löst die MMU einen Trap aus.
- ○ Das Betriebssystem erkennt die ungültige Adresse vor Ausführung eines Maschinenbefehls und lagert die Seite zuerst ein bevor ein Fehler passiert.
- ○ Die MMU erkennt bei der Adressumsetzung, dass die physikalische Adresse ungültig ist und löst einen Trap aus.

不是发生在 adresseumsetzung 的时候，而是 发生在 program 执行的时候，发现 在 page table 里面的 presence bit 是 0 的时候，会 触发 demanding page。

Die virtuell adresse wird von MMU in die physikalische adresse umgesetzt, wenn eine seite ausgelagert , dann in der tabelle gibt es ein bit,

-----> page fault ----> demanding page, diese page wird noch mal eingelagert ----> tabelle bit wird geandert und program 第二次 被尝试 执行。

b) Welche der folgenden Aussagen über Schedulingverfahren ist richtig? — 2 Punkte

- ○ Bei preemptivem Scheduling sind Prozessumschaltungen unmöglich, wenn ein Prozess in einer Endlosschleife läuft.
- ○ Bei kooperativem Scheduling sind Prozessumschaltungen unmöglich wenn ein Prozess in einer Endlosschleife läuft, selbst wenn er bei jedem Schleifendurchlauf einen Systemaufruf macht.
- ○ Preemptives Scheduling ist für Mehrbenutzerbetrieb geeignet.
- ○ Kooperatives Scheduling ist für Steuerungssysteme mit Echtzeitanforderungen völlig ungeeignet.

the cpu will kick this process out

selbst entscheidet wann verlassen

jeder Benutzer hat einige "time slice"

有一些 比较 重要的 process, , 你就得等着 这个 process 完事，不能 interrupt

搞清 preemptive scheduling 和 kooperative scheduling 的 区别:

Process Dispatching: Time and Selection

- transitions to the ready state (READY) update the CPU ready list
  - a decision regarding the **placement** of the **process control block** is made
  - the result depends on the system's **CPU scheduling strategy**
- scheduling or rescheduling (dt. Einplanung/Umplanung) occurs, ...
  1. after a process has been created
  2. when a process yields control of the CPU
  3. if the event which is expected by a process has occurred
  4. as soon as an swapped-out process is resumed

A process can be **forced** to release the CPU → preemptive scheduling

- for example, by a timer interrupt ②

CPU entscheidet, process weg oder nicht
OS kann den Process entziehen

Cooperative scheduling **is a style of scheduling in which the OS never interrupts a running process to initiate a context switch from one process to another**. Processes must voluntarily yield control periodically or when logically blocked on a resource.

process entscheidet selbst, wann weg

bei preemptive Scheduling is es meistens mit "time slice " zusammen

time slice 用完之后，egal process fertig oder nicht， weg!

Time-slice Driven: Round Robin (RR)

- reduces the **penalization of processes with short CPU bursts** that occurs with FCFS:
  - the **processor time** is divided into **time slices** (dt. Zeitscheiben)
- when the time slice expires, a process change may take place
  - the interrupted process is pushed to the end of the ready list
  - the next process is taken from the ready list according to FCFS
- basis for CPU protection: timer interrupt enforces end of time slice
- the **time slice length** determines the effectiveness of this scheduling method
  - time slice
    - too long: degeneration of RR to FCFS
    - too short: high scheduling overhead
  - rule of thumb: slightly longer than the duration of a "typical interaction"

1. The basic difference between preemptive and non-preemptive scheduling is that in preemptive scheduling the CPU is allocated to the processes for the **limited** time. While in Non-preemptive scheduling, the CPU is allocated to the process till it **terminates** or switches to **waiting state**.
2. The executing process in preemptive scheduling is **interrupted** in the middle of execution whereas, the executing process in non-preemptive scheduling is **not interrupted** in the middle of execution.
3. Preemptive Scheduling has the **overhead** of switching the process from ready state to running state, vise-verse, and maintaining the ready queue. On the other hands, non-preemptive scheduling has **no overhead** of switching the process from running state to ready state.

c) Welche der folgenden Aussagen zum Thema „Aktives Warten" ist richtig? 2 Punkte

浪费

- ○ Aktives Warten vergeudet gegenüber passivem Warten immer CPU-Zeit.
- ○ Bei verdrängenden Scheduling-Strategien verzögert aktives Warten nur den betroffenen Prozess, behindert aber nicht andere.
- ○ Aktives Warten auf andere Prozesse darf bei nicht-verdrängenden Scheduling-Strategien auf einem Monoprozessorsystem nicht verwendet werden.
- ○ Auf Mehrprozessorsystemen ist aktives Warten unproblematisch und deshalb dem passiven Warten immer vorzuziehen.

你占用 cpu 进行 dead loop， 其他 procss 就 无法有 足够的 cpu 资源了

单声道的

更好的

wenn das Warten sehr lange dauert, dann passive warten ist ok(denn不会占用CPU)
相反 aktive warten 会占用 很多CPU， 所以 不行

1:

kurze zeit "aktive warten" ist manchmal effizienter als "passive warten"
也叫 busy waiting.
因为 passive waiting 涉及到添加到queue，当需要很短的时间的时候， aktive waiting 会 用 更短的时间

3:
当一个 cpu 只能 处理 一个 process 的时候（monoprocessorsystem）， und diese Process kann nicht verdraegend werden(da unter cooperative scheduling)
, und gleichzeitig foedert dise Process einen anderen Process, ----> dead lock

抵押

d) Welche Seitennummer (*page number*) und welcher Versatz (*offset*) gehören bei einstufiger Seitennummerierung und einer Seitengröße von 2048 Bytes zu folgender logischer Adresse: 0xba1d 2 Punkte

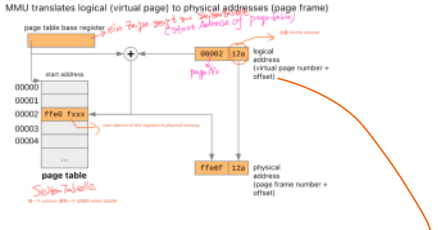
- ○ Seitennummer 0xb, Versatz 0xa1d
- ○ Seitennummer 0x17, Versatz 0x21d
- ○ Seitennummer 0x2e, Versatz 0x21d
- ○ Seitennummer 0xba, Versatz 0x1d

Memory Management - Segmentation

- map logical to physical addresses with hardware support

Memory Management – Paging

- MMU translates logical (virtual page) to physical addresses (page frame)



0xba1d = 0b 1011 1010 0001 1101

2048 = 2^11 ， 说明 memory 需要占用 11 个 bits

0xba1d = 0b 1011 1010 0001 1101

2048 = 2^11 ， 说明 memory 需要占用 11 个 bits

那就把 0xba1d 划分一下： 1011 1[illegible]

绿色是： memory offset： 0x21d
黄色是： page number： 0x17

2048byte 就是 在指导你： wie viele 位 brauchst du， 来记录 2048.

问题

e) Welche Problematik kann durch das Philosophenproblem beschrieben werden? 2 Punkte

- ○ Ein Erzeuger und ein Verbraucher greifen gleichzeitig auf gemeinsame Datenstrukturen zu.
- ○ Exklusive Bearbeitung durch mehrere Bearbeitungsstationen.
- ○ Potenzielle Verklemmung durch eine ungünstige Anforderungsreihenfolge geteilter Betriebsmittel durch mehrere Prozesse.
- ○ Mehrere Prozesse greifen lesend und schreibend auf gemeinsame Datenstrukturen zu.

f) Welche Aussage über den Rückgabewert von `fork(2)` ist richtig? 2 Punkte

- ○ Der Rückgabewert ist in jedem Prozess (Kind und Vater) jeweils die eigene Prozess-ID.
- ○ Der Kind-Prozess bekommt die Prozess-ID des Vater-Prozesses.
- ○ Im Fehlerfall wird im Kind-Prozess -1 zurückgeliefert.

im Fehler fall wird kein Kind erzeugt und einfach -1 returned

- ○ Dem Vater-Prozess wird die Prozess-ID des Kind-Prozesses zurückgeliefert.

```
int a = 5;
pid_t p = fork(); // (1)
a += p; // (2)
if (p == -1) {
  // fork-Fehler
  //es wurde kein Kind erzeugt
  ...
} else if (p == 0) {
  // Kindprozess
} else {
  // Elternprozess
  // p ist die PID des neu
  // erzeugten Kindprozesses
  ...
}
```

在 parent 里面的p 的值 就是 kinder process id

Upon successful completion, fork() returns a value of 0to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1is returned to the parent process, no child process is created, and the global variable errno is setto indicate the error.

g) Welche Aussage über Funktionen der `exec(3)`-Familie ist richtig? 2 Punkte

- ○ Dem Vater-Prozess wird die Prozess-ID des neu erzeugten Kind-Prozesses zurückgeliefert.
- ○ Beim Aufruf von `exec()` wird das im aktuellen Prozess laufende Programm durch das angegebene Programm ersetzt.

kannst du nicht mehr die Adresseraum zugreifen, 因为 被新的 运行程序 取代了

- ○ Nach einem erfolgreichen Aufruf von `exec()` kann weiterhin auf Datenstrukturen im Adressraum des Aufrufers zugegriffen werden.
- ○ Der an `exec()` übergebene Funktionszeiger wird durch einen neuen Thread im aktuellen Prozess ausgeführt.

不会，会 截至当前的进程，把 当前的进程 换成自己想要 运行的程序

```
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
```

char* arg1 , char* arg2, 就是 字符串一个一个地给

区别

char* const argv[] 就是 一个 const 的数组，里面盛放的是 字符串

- Lädt Programm zur Ausführung in den aktuellen Prozess
  - aktuell ausgeführtes Programm wird ersetzt (Text-, Daten- und Stacksegment)
  - erhalten bleiben: Dateideskriptoren (= geöffnete Dateien), Arbeitsverzeichnis, ...
- Aufrufparameter für exec(3)
  - Dateiname des neuen Programmes
  - Argumente, die der main-Funktion des neuen Programms übergeben werden

就是 命令行的 命令

- exec kehrt nur im Fehlerfall zurück

没有返回值，所以不用überprufen

```
int val;
val =execlp("ls", "ls","-l", NULL);
if(val == -1)
    perror("execl error");
```

```
int main(void)
{
    printf("entering main process---\n");
    int ret;
    char *argv[] = {"ls","-l",NULL};
    ret = execvp("ls",argv);
    if(ret == -1)
        perror("execl error");
    printf("exiting main process ----\n");
    return 0;
}
```

C语言中定义一个 数组， 里面盛放的是字符串

char* argv[] = {"ls", "-l", NULL};

这里面的 argv[0] = "ls"

h) Welche Aussage über Schedulingverfahren ist richtig? 2 Punkte

垄断 要是 线程 自己一直不结束, und immer noch kein system aufruf macht, dann 这个process 就垄断了 CPU

- Bei kooperativem Scheduling kann es zur Monopolisierung der CPU kommen.
- Round-Robin bevorzugt E/A-intensive Prozesse zu Gunsten von rechenintensiven Prozessen.

偏爱

- Der Konvoieffekt kann bei kooperativen Schedulingverfahren wie First-Come-First-Served nicht auftreten.

一个大车 满， 后面的小车 都得等

- Beim Einsatz preemptiver Schedulingverfahren kann laufenden Prozessen die CPU nicht entzogen werden.

doch! OS 可以随时剥夺 CPU 的 使用权

Time-slice Driven: Round Robin (RR)

- reduces the **penalization of processes with short CPU bursts** that occurs with FCFS:
  - the processor time is divided into time slices (dt. Zeitscheiben)
- when the time slice expires, a process change may take place
  - the interrupted process is pushed to the end of the ready list
  - the next process is taken from the ready list according to FCFS
- basis for CPU protection: timer interrupt enforces end of time slice
- the **time slice length** determines the effectiveness of this scheduling method
  - time slice
    - too long: degeneration of RR to FCFS
    - too short: high scheduling overhead
  - rule of thumb: slightly longer than the duration of a "typical interaction"

Discussion: Round Robin – Performance Issues

- **I/O-heavy** processes finish CPU burst within their time slice
  - they block and return to the ready list at the end of their I/O burst
  - it is likely that their time slice has not yet been exhausted
- **CPU-heavy** processes, on the other hand, make full use of their time slice
  - they are preempted and immediately return to the ready list
- result: **CPU time is unevenly distributed** in favor of CPU-heavy processes
  - causal link: I/O-heavy processes are poorly operated and thus devices poorly utilized
  - response time of I/O-heavy processes increases

Time-slice Driven: Virtual Round Robin (VRR)

- VRR addresses the uneven distribution of CPU times that is possible with RR
  - processes are added to a *preferred list* at the end of their I/O bursts
  - scheduler processes *preferred list* **before** the *ready list*
- the method works with **time slices of variable lengths**
  - processes on the *preferred list* are **not** allocated a full time slice
  - instead, they are granted the remaining term of their previously not fully used time
  - if their CPU burst takes longer, they are preempted and put back to the ready list
- process handling more complex compared to RR → overhea

KONVOI-EFFEKT IN BETRIEBSSYSTEMEN

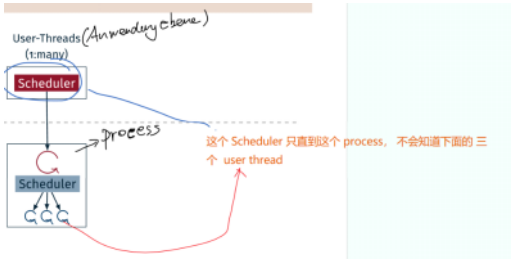
Voraussetzungen: Grundlagen des FCFS-Scheduling (Programm für FCFS-Scheduling | Set 1 , Programm für FCFS-Scheduling | Set 2)

Der Konvoi-Effekt ist ein Phänomen im Zusammenhang mit dem First Come First Serve (FCFS)-Algorithmus, bei dem das gesamte Betriebssystem aufgrund weniger langsamer Prozesse langsamer wird.

Figure - The Convey Effect, Visualized

Discussion: FCFS – The Convoy Effect

- the problem is faced by **short-running I/O-heavy processes** that follow long-running **CPU-heavy processes**
  - processes with **long CPU bursts** are **favored**
  - processes with **short CPU bursts** are **penalized**
- FCFS **minimizes** the number of **context switches**. However the convoy effect causes the following problems:
  - high response times
  - low I/O throughput
- FCFS is therefore unsuitable for a mixed workloads (CPU- and I/O-heavy processes)
- typically, FCFS is only in used in batch processing systems



i) Welche der folgenden Aussagen zum Thema Threads und Prozesse ist richtig?

2 Punkte

- ○ Zu jedem Thread (*Light-weight Process*) gehört ein eigener isolierter Adressraum.
- ○ Threads (*Light-weight Processes*) teilen sich den kompletten Adressraum und verwenden daher den selben Stack.
- ✓ User-level Threads (*Feather-weight Processes*) blockieren sich bei blockierenden Systemaufrufen gegenseitig.
- ○ Die Umschaltung von User-level Threads (*Feather-weight Processes*) ist eine privilegierte Operation und muss deshalb im Systemkern erfolgen.

user thread blokieren 说明 承载 user thread 的process（这个process 负责把自己的 算力 和资源分配给 user threads ）被 blokiert 了，所以 所有的 user thread 都被blokiert了，因为大家都在一条船上。

das ist user thread, und es sollte in user space erfolgen, es ist nicht in Systemkern, und system kern 看不见 user thread

Federgewichtige Prozesse (User-Threads)

das heisst, das Betriebssystem sieht dieses Thread nicht, das Betriebssystem sieht nur den Konkrollfluss, der dieses Thread implementiert.

- Realisierung auf Anwendungsebene
- Systemkern sieht nur **einen** Kontrollfluss
- \+ Erzeugung von Threads extrem billig
- − Systemkern hat kein Wissen über diese Threads
  - in Multiprozessorsystemen keine parallelen Abläufe möglich
  - wird **ein** User-Thread blockiert, sind **alle** User-Threads blockiert
  - Scheduling zwischen den Threads schwierig

der Process, in den das "user thread" erzeugt wird, gibt das user thread ein bisschen Zeit slot, so scheint es dass es parallel ist.

Leichtgewichtige Prozesse (Kernel-Threads)

gemeinsam Adressraum

Kernel Thread ist was wir unter "Thread" versteht, 我们使用的 就是这个

- \+ Gruppe von Threads nutzt gemeinsam die Betriebsmittel eines Prozesses
- \+ jeder Thread ist als eigener Aktivitätsträger dem Betriebssystemkern bekannt
- − Kosten für Erzeugung erheblich geringer als bei Prozessen, aber erheblich teuerer als bei User-Threads

j) Was versteht man unter der Second-Chance- (oder Clock-) Policy?

2 Punkte

- ○ Eine Seitenersetzungsstrategie, bei der jeweils die älteste Seite ausgelagert wird.
- ○ Eine Speicherallokationsstrategie, bei der im Fehlerfall ein zweiter Allokationsversuch stattfindet.
- ✓ Eine Seitenersetzungsstrategie, die mit Hilfe eines Referenz-Bits eine einfacher zu implementierende Annäherung an LRU realisiert.
- ○ Eine Scheduling-Strategie, bei der Prozesse vor der Verdrängung eine zweite Chance erhalten.

second chance is in "virtual memory"

就是 speicher 不够，把 谁 swap out 出去

k) Was versteht man unter virtuellem Speicher?

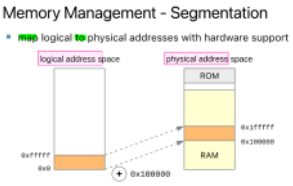
k) Was versteht man unter virtuellem Speicher?

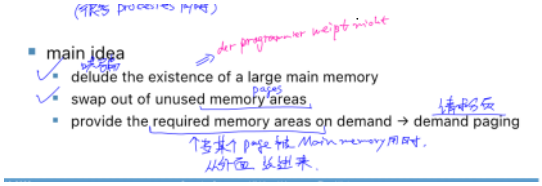
2 Punkte

- ○ Speicher, der nur im Betriebssystem sichtbar ist, jedoch nicht für einen Anwendungsprozess.
- ○ Unter einem virtuellen Speicher versteht man einen physikalischen Adressraum, dessen Adressen durch eine MMU vor dem Zugriff auf logische Adressen umgesetzt werden.
- ○ Adressierbarer Speicher in dem sich keine Daten speichern lassen, weil er physikalisch nicht vorhanden ist.
- ○ Speicher, der einem Prozess durch entsprechende Hardware (MMU) und durch Ein- und Auslagern von Speicherbereichen vorgespiegelt wird, aber möglicherweise größer als der verfügbare physikalische Hauptspeicher ist.

就是 Speicher durch ... durch ... 把 process 欺骗了，

伴装 欺骗





2) Mehrfachauswahlfragen (8 Punkte)

Bei den Mehrfachauswahlfragen in dieser Aufgabe sind jeweils m Aussagen angegeben, davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie alle richtigen Aussagen an.
Jede korrekte Antwort in einer Teilaufgabe gibt einen Punkt, jede falsche Antwort einen Minuspunkt.
Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, d. h. falsche Antworten wirken sich nicht auf andere Teilaufgaben aus.

Wollen Sie eine falsch angekreuzte Antwort korrigieren, streichen Sie bitte das Kreuz mit drei waagrechten Strichen durch (☒).

Lesen Sie die Frage genau, bevor Sie antworten.

a) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet sind richtig?

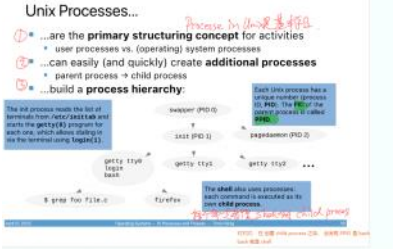
4 Punkte

- ☐ UNIX-Prozesse sind hierarchisch organisiert. tree structure, parents process, child process
- ☐ Der UNIX-Systemaufruf `fork(2)` lädt eine Programmdatei in einen neu erzeugten Prozess. one program may have many process
- ☐ Ein Programm kann durch mehrere Prozesse gleichzeitig ausgeführt werden.
- ☐ Der UNIX-Systemaufruf `fork(2)` erzeugt, von wenigen Aspekten abgesehen, eine Kopie des aufrufenden Prozess. 这是属于 memory 和 OS 的， Program 只是 用， 并不拥有。
- ☐ Das Programm ist der statische Teil (Rechte, Speicher, etc.), der Prozess der aktive Teil (Programmzähler, Register, Stack). 属于 cpu
- ☐ Ein Prozess kann mithilfe von Threads mehrere Programme gleichzeitig ausführen. process一次只能 服务于 一个process
- ☐ Ein Prozess ist ein Programm in Ausführung - ein Prozess kann während seiner Lebenszeit aber auch mehrere verschiedene Programme ausführen.
- ☐ Der Compiler erzeugt aus einer oder mehreren Objekt-Dateien (Modulen) einen Prozess. 这里 Compiler 应该 创建 program（可执行.exe 文件）， 不是 prozess


Program Counter is a register in the CPU hardware. Effectively it's a digital counter as consists of binary latches where each latch represents a binary bit. Number of latches,is the size of the PC depends on the processor architecture. 15 Jul 2015

Where is Program Counter (PC) stored?

a program can load and run another program within the same process. So a process can execute several programs, but at any given moment in times it's only one active.



A program is a passive entity that contains the set of codes required to perform a certain task. A process is an active instance of the program which is started when the program is executed. Once a program is executed, a process is started by the program. The process executes the instructions written in the program. 30 Jun 2022

https://www.scaler.com › topics › difference-between-pro...

Difference between Process and Program | Scaler Topics

Unix Processes in detail: `fork()`

System Call: `pid_t fork (void)`

- duplicates the running process (**process creation**)
- the child process inherits:
  - address space (`code, data, bss, heap, stack`)
  - user ID
  - standard I/O channels
  - process group, signal table (more about this later)
  - open files, current working directory (more on this *much* later).
- what is not copied:
  - process ID (PID), parent process ID (PPID)
  - pending signals, accounting data, ...

one process calls fork, but two processes return...

Discussion: Fast Process Creation

- copying the address space takes a lot of time
- historical solution: `vfork()`
  - parent process is suspended until the child process calls `exec..()` or terminates with `_exit()`
  - the child process simply uses code and data from the parent process (zero copying!)
  - the child process must not modify any data
    - sometimes not so easy: for example, do not call `exit()`, but `_exit()`!
- today: copy-on-write (COW)
  - with the help of the MMU, parent and child process share the same code and data segment
  - only when the child process changes data, the segment is copied
  - if `fork()` is followed directly by an `exec..()`, this does not occur
  - `fork()` with COW is hardly slower than `vfork()`

signal 并不总是 kritische problems

b) Welche der folgenden Aussagen zum Thema UNIX-Signale sind richtig?

4 Punkte

- ☐ UNIX-Signale sind stets ein Hinweis auf ein kritisches Problem, das zwingend zur Beendigung des aktuell laufenden Programms führen muss.
- ☐ Durch Signale können Nebenläufigkeitsprobleme in grundsätzlich nicht-parallelen Programmen entstehen.
- ☐ Signale können dazu führen, dass blockierende Systemaufrufe mit der errno EINTR abgebrochen werden.
- ☐ Signale haben keine praktische Relevanz, da neben der Signalnummer keinerlei weitere Nutzinformation übermittelt werden kann.
- ☐ Geräte nutzen UNIX-Signale, um der aktuell laufenden Anwendung oder dem Betriebssystem das Vorliegen neuer Ereignisse mitzuteilen.
- ☐ Programme können für die meisten Signale eine eigene Funktionen zur deren Behandlung bereitstellen.
- ☐ UNIX erlaubt es nicht, Signale an blockierte Prozesse zuzustellen, da dies dazu führen würde, dass wichtige Systemaufrufe unterbrochen würden.
- ☐ Signale, die an bereite, aber nicht laufende Prozesse zugestellt werden, werden abgearbeitet sobald der Prozess die CPU zugeteilt bekommt.

Was sind nebenläufige Prozesse?

Zwei Vorgänge oder **Prozesse** A und B heißen **nebenläufig**, wenn sie voneinander unabhängig bearbeitet werden können. Dabei ist es egal, ob zuerst der Vorgang A und dann B ausgeführt wird, oder ob sie in umgekehrter Reihenfolge abgearbeitet werden oder ob sie gleichzeitig erledigt werden.

即便使是 nicht parallelen Program, 但是 一个process执行的过程中， signal 来 带来了 最新的 information， 就会 产生 一半 是 new informatin, 一半是 old information

等价于 一个 blokiert 的 Process

比如说，一个 systemaufruf(Process) 正在 blockiert 的 状态， dann kommt ein signal，dann 不再是 block 的状态了，进入ready 的 状态

Gerät nutzen Interrupt

Geraet geben 最新结果 给 process via interrupt，比如 点击 鼠标

What is interrupt example in OS?

For example, pressing a keyboard key or moving a mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position. Hardware interrupts can arrive asynchronously for the processor clock and at any time during instruction execution.

再 Vorlesung 11 里面

Signals

- signals are interrupts recreated in software
  - similar to those of a processor through I/O devices
  - **minimal form** of inter-process communication (transmission of the signal number, **no payload**)
- **sender:**
  - processes - with the help of the system call `kill(2)`
  - operating system - when certain events occur
- **receiver** process performs signal handling:
  - ignore
  - terminate process
  - call a signal handler function
    - after handling the signal, the process continues at the interrupted positior

sigmal就是 process 之间 进行通信的。

Signals are similar to interrupts, the difference being that **interrupts are mediated by the CPU and handled by the kernel while signals are mediated by the kernel (possibly via system calls) and handled by individual processes**.

Signals

- with the help of signals, processes can be informed about exceptional situations (c.f., hardware interrupts)
- examples:
  - **SIGINT** abort process (e.g., Ctrl-C)
  - **SIGSTOP** stop process (e.g., Ctrl-Z)
  - **SIGCHLD** child process terminated
  - **SIGSEGV** memory protection violation of the process
  - **SIGKILL** process is killed
- the default signal handling (terminate, stop, ...) can be redefined for most signals
  - see **signal**(7)

- the default signal handling (terminate, stop, ...) can be redefined for most signals
  - see **`signal`**`(7)`

Signals – Technical View

- signal handling always takes place at the transition from kernel to user mode
- What happens when the receiving process...
  1. runs in state **RUNNING** (e.g., segmentation error, bus error)?
     - immediate start of the signal handling routine
  2. does not run but is in state **READY** (e.g., `kill` system call)?
     - the signal is registered in the process control block (PCB)
     - as soon as the process runs again, the signal handling takes place
  3. waits on an I/O event, in state **BLOCKED**?
     - the I/O system call (e.g., `read`) is aborted with `EINTR`
     - the process state is set to **READY**
     - after that: as for 2.
     - if applicable, the interrupted system call is executed again (SA_RESTART)

11111

Aufgabe 1: Ankreuzfragen (30 Punkte)

1) Einfachauswahlfragen (22 Punkte)

Bei den Einfachauswahlfragen in dieser Aufgabe ist jeweils nur **eine** richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Antwort korrigieren, streichen Sie bitte die falsche Antwort mit drei waagrechten Strichen durch (⊗) und kreuzen die richtige an.

Lesen Sie die Frage genau, bevor Sie antworten.

a) Wie wird erkannt, dass eine Seite eines virtuellen Adressraums gerade ausgelagert ist? 2 Punkte

- ○ Bei Programmen, die in virtuellen Adressräumen ausgeführt werden sollen, erzeugt der Compiler speziellen Code, der vor Betreten einer Seite die Anwesenheit überprüft und ggf. die Einlagerung veranlasst.
- ○ Im Seitendeskriptor wird ein spezielles Bit geführt, das der MMU zeigt, ob eine Seite eingelagert ist oder nicht. Falls die Seite nicht eingelagert ist, löst die MMU einen Trap aus.
- ○ Das Betriebssystem erkennt die ungültige Adresse vor Ausführung eines Maschinenbefehls und lagert die Seite zuerst ein bevor ein Fehler passiert.
- ○ Die MMU erkennt bei der Adressumsetzung, dass die physikalische Adresse ungültig ist und löst einen Trap aus.

b) Welche der folgenden Aussagen über Schedulingverfahren ist richtig? 2 Punkte

- ○ Bei preemptivem Scheduling sind Prozessumschaltungen unmöglich, wenn ein Prozess in einer Endlosschleife läuft.
- ○ Bei kooperativem Scheduling sind Prozessumschaltungen unmöglich wenn ein Prozess in einer Endlosschleife läuft, selbst wenn er bei jedem Schleifendurchlauf einen Systemaufruf macht.
- ○ Preemptives Scheduling ist für Mehrbenutzerbetrieb geeignet.
- ○ Kooperatives Scheduling ist für Steuerungssysteme mit Echtzeitanforderungen völlig ungeeignet.

c) Welche der folgenden Aussagen zum Thema „Aktives Warten“ ist richtig? 2 Punkte

- ○ Aktives Warten vergeudet gegenüber passivem Warten immer CPU-Zeit.
- ○ Bei verdrängenden Scheduling-Strategien verzögert aktives Warten nur den betroffenen Prozess, behindert aber nicht andere.
- ○ Aktives Warten auf andere Prozesse darf bei nicht-verdrängenden Scheduling-Strategien auf einem Monoprozessorsystem nicht verwendet werden.
- ○ Auf Mehrprozessorsystemen ist aktives Warten unproblematisch und deshalb dem passiven Warten immer vorzuziehen.

d) Welche Seitennummer (*page number*) und welcher Versatz (*offset*) gehören bei einstufiger Seitennummerierung und einer Seitengröße von 2048 Bytes zu folgender logischer Adresse: 0xba1d 2 Punkte

- ○ Seitennummer 0xb, Versatz 0xa1d
- ○ Seitennummer 0x17, Versatz 0x21d
- ○ Seitennummer 0x2e, Versatz 0x21d
- ○ Seitennummer 0xba, Versatz 0x1d

e) Welche Problematik kann durch das Philosophenproblem beschrieben werden? 2 Punkte

- ○ Ein Erzeuger und ein Verbraucher greifen gleichzeitig auf gemeinsame Datenstrukturen zu.
- ○ Exklusive Bearbeitung durch mehrere Bearbeitungsstationen.
- ○ Potenzielle Verklemmung durch eine ungünstige Anforderungsreihenfolge geteilter Betriebsmittel durch mehrere Prozesse.
- ○ Mehrere Prozesse greifen lesend und schreibend auf gemeinsame Datenstrukturen zu.

f) Welche Aussage über den Rückgabewert von `fork(2)` ist richtig? 2 Punkte

- ○ Der Rückgabewert ist in jedem Prozess (Kind und Vater) jeweils die eigene Prozess-ID.
- ○ Der Kind-Prozess bekommt die Prozess-ID des Vater-Prozesses.
- ○ Im Fehlerfall wird im Kind-Prozess -1 zurückgeliefert.
- ○ Dem Vater-Prozess wird die Prozess-ID des Kind-Prozesses zurückgeliefert.

g) Welche Aussage über Funktionen der `exec(3)`-Familie ist richtig? 2 Punkte

- ○ Dem Vater-Prozess wird die Prozess-ID des neu erzeugten Kind-Prozesses zurückgeliefert.
- ○ Beim Aufruf von `exec()` wird das im aktuellen Prozess laufende Programm durch das angegebene Programm ersetzt.
- ○ Nach einem erfolgreichen Aufruf von `exec()` kann weiterhin auf Datenstrukturen im Adressraum des Aufrufers zugegriffen werden.
- ○ Der an `exec()` übergebene Funktionszeiger wird durch einen neuen Thread im aktuellen Prozess ausgeführt.

h) Welche Aussage über Schedulingverfahren ist richtig?

2 Punkte

- ○ Bei kooperativem Scheduling kann es zur Monopolisierung der CPU kommen.
- ○ Round-Robin bevorzugt E/A-intensive Prozesse zu Gunsten von rechenintensiven Prozessen.
- ○ Der Konvoieffekt kann bei kooperativen Schedulingverfahren wie First-Come-First-Served nicht auftreten.
- ○ Beim Einsatz preemptiver Schedulingverfahren kann laufenden Prozessen die CPU nicht entzogen werden.

i) Welche der folgenden Aussagen zum Thema Threads und Prozesse ist richtig?

2 Punkte

- ○ Zu jedem Thread (*Light-weight Process*) gehört ein eigener isolierter Adressraum.
- ○ Threads (*Light-weight Processes*) teilen sich den kompletten Adressraum und verwenden daher den selben Stack.
- ○ User-level Threads (*Feather-weight Processes*) blockieren sich bei blockierenden Systemaufrufen gegenseitig.
- ○ Die Umschaltung von User-level Threads (*Feather-weight Processes*) ist eine privilegierte Operation und muss deshalb im Systemkern erfolgen.

j) Was versteht man unter der Second-Chance- (oder Clock-) Policy?

2 Punkte

- ○ Eine Seitenersetzungsstrategie, bei der jeweils die älteste Seite ausgelagert wird.
- ○ Eine Speicherallokationsstrategie, bei der im Fehlerfall ein zweiter Allokationsversuch stattfindet.
- ○ Eine Seitenersetzungsstrategie, die mit Hilfe eines Referenz-Bits eine einfacher zu implementierende Annäherung an LRU realisiert.
- ○ Eine Scheduling-Strategie, bei der Prozesse vor der Verdrängung eine zweite Chance erhalten.

k) Was versteht man unter virtuellem Speicher?

2 Punkte

- ○ Speicher, der nur im Betriebssystem sichtbar ist, jedoch nicht für einen Anwendungsprozess.
- ○ Unter einem virtuellen Speicher versteht man einen physikalischen Adressraum, dessen Adressen durch eine MMU vor dem Zugriff auf logische Adressen umgesetzt werden.
- ○ Adressierbarer Speicher in dem sich keine Daten speichern lassen, weil er physikalisch nicht vorhanden ist.
- ○ Speicher, der einem Prozess durch entsprechende Hardware (MMU) und durch Ein- und Auslagern von Speicherbereichen vorgespiegelt wird, aber möglicherweise größer als der verfügbare physikalische Hauptspeicher ist.

2) Mehrfachauswahlfragen (8 Punkte)

Bei den Mehrfachauswahlfragen in dieser Aufgabe sind jeweils m Aussagen angegeben, davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie alle richtigen Aussagen an.
Jede korrekte Antwort in einer Teilaufgabe gibt einen Punkt, jede falsche Antwort einen Minuspunkt. Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, d. h. falsche Antworten wirken sich nicht auf andere Teilaufgaben aus.

Wollen Sie eine falsch angekreuzte Antwort korrigieren, streichen Sie bitte das Kreuz mit drei waagrechten Strichen durch (☒).

Lesen Sie die Frage genau, bevor Sie antworten.

a) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet sind richtig?

4 Punkte

- ☐ UNIX-Prozesse sind hierarchisch organisiert.
- ☐ Der UNIX-Systemaufruf `fork(2)` lädt eine Programmdatei in einen neu erzeugten Prozess.
- ☐ Ein Programm kann durch mehrere Prozesse gleichzeitig ausgeführt werden.
- ☐ Der UNIX-Systemaufruf `fork(2)` erzeugt, von wenigen Aspekten abgesehen, eine Kopie des aufrufenden Prozess.
- ☐ Das Programm ist der statische Teil (Rechte, Speicher, etc.), der Prozess der aktive Teil (Programmzähler, Register, Stack).
- ☐ Ein Prozess kann mithilfe von Threads mehrere Programme gleichzeitig ausführen.
- ☐ Ein Prozess ist ein Programm in Ausführung - ein Prozess kann während seiner Lebenszeit aber auch mehrere verschiedene Programme ausführen.
- ☐ Der Compiler erzeugt aus einer oder mehreren Objekt-Dateien (Modulen) einen Prozess.

b) Welche der folgenden Aussagen zum Thema UNIX-Signale sind richtig?

4 Punkte

- ☐ UNIX-Signale sind stets ein Hinweis auf ein kritisches Problem, das zwingend zur Beendigung des aktuell laufenden Programms führen muss.
- ☐ Durch Signale können Nebenläufigkeitsprobleme in grundsätzlich nicht-parallelen Programmen entstehen.
- ☐ Signale können dazu führen, dass blockierende Systemaufrufe mit der errno `EINTR` abgebrochen werden.
- ☐ Signale haben keine praktische Relevanz, da neben der Signalnummer keinerlei weitere Nutzinformation übermittelt werden kann.
- ☐ Geräte nutzen UNIX-Signale, um der aktuell laufenden Anwendung oder dem Betriebssystem das Vorliegen neuer Ereignisse mitzuteilen.
- ☐ Programme können für die meisten Signale eine eigene Funktionen zur deren Behandlung bereitstellen.
- ☐ UNIX erlaubt es nicht, Signale an blockierte Prozesse zuzustellen, da dies dazu führen würde, dass wichtige Systemaufrufe unterbrochen würden.
- ☐ Signale, die an bereite, aber nicht laufende Prozesse zugestellt werden, werden abgearbeitet sobald der Prozess die CPU zugeteilt bekommt.

Aufgabe 2: parrots (60 Punkte)

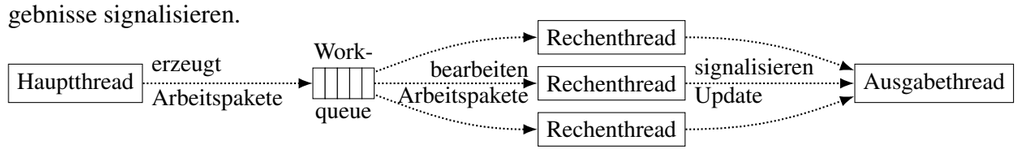
Sie dürfen diese Seite zur besseren Übersicht bei der Programmierung heraustrennen!

Schreiben Sie ein Programm `parrots` (**par**allel **ro**w-based **t**riangle **s**ummer), das zeilenweise Koordinaten von Dreiecken aus einer per Befehlszeilenargument übergebenen Datei einliest und die Anzahl der ganzzahligen Koordinaten auf den Kanten und innerhalb der Dreiecke berechnet.

Beispielhafter Aufruf von `parrots`:

```
chris@host:~$ ./parrots triangles.txt
```

`parrots` liest zeilenweise Dreiecke im Format *(x1,y1),(x2,y2),(x3,y3)* ein. Zeilen, die eine maximale Länge von 1024 Zeichen (`MAX_LINE`, exklusive `'\n'/'\0'`) überschreiten, oder nicht dem erwarteten Format entsprechen, werden dabei unter Ausgabe einer entsprechenden Warnmeldung ignoriert. Die Funktionen zur Umwandlung der eingelesen Zeile in ein **`struct triangle`** und zur Berechnung der Punkte sind im Modul `triangle.o` **vorgegebenen** (siehe `parseTriangle()` und `countPoints()` in der angehängte Manpage `triangle(3)`).
Der aktuelle Zwischenstand der Berechnungen wird regelmäßig aktualisiert und ausgegeben.
Zur Steigerung der Berechnungsgeschwindigkeit werden die Berechnungen an `CALC_THREADS` Rechenthreads ausgelagert, die einem dedizierten Ausgabethread das Vorliegen neuer Zwischenergebnisse signalisieren.



Struktur des Programms:

- Der *Hauptthread* initialisiert die benötigten Datenstrukturen und erzeugt `CALC_THREADS` Rechenthreads und den einen Ausgabethread. Dann liest er zeilenweise die per Befehlszeilenargument übergebene Datei ein, erzeugt Arbeitspakete in Form von **`struct triangle`**s und fügt die Arbeitspakete in die Workqueue ein. Nachdem alle Dreiecke eingelesen und abgearbeitet wurden, wartet er auf die Beendigung **aller** Threads, gibt alle allokierte Ressourcen frei und beendet sich.
- Die *Rechenthreads* (in obiger Illustration: 3), entnehmen jeweils Dreiecke aus der Workqueue und führen die eigentliche Zählung der Punkte mittels der vorgegebenen Funktion `countPoints()` durch. Sobald `countPoints()` die berechneten Werte liefert, signalisiert der Rechenthread dem Ausgabethread das Vorliegen neuer Werte.
- Der *Ausgabethread* gibt nach Signalisierung durch die Rechenthreads die akkummulierte Anzahl aller *interior* und *boundary* Koordinaten (siehe `triangle(3)`) auf `stdout` aus.

Die Kommunikation zwischen den Threads soll mittels modulglobaler Variablen geschehen. Es ist keine Fehlerbehandlung für die Ausgabe auf `stdout` und `stderr` nötig.

Ein zentraler Bestandteil dieser Aufgabe ist die **korrekte Synchronisation** mithilfe der vorgegebenen, **passiv wartenden Semaphor-Implementierung** – aktives Warten ist nicht erlaubt. Langsame Funktionen (z.B. `printf(3)`) dürfen nicht in kritischen Abschnitten ausgeführt werden.

Hinweise:

- Speichern Sie den Wert `NULL` in der Queue um die Arbeiterthreads zuverlässig zu beenden.
- Gehen Sie davon aus, dass alle Operationen der Queue ohne weitere Synchronisation nebenläufig genutzt werden können.
- Ihnen steht das aus der Übung bekannte Semaphor-Modul zur Verfügung. Die Schnittstelle finden Sie im folgenden Programmgerüst nach den **`#include`**-Anweisungen.
- Eine Beschreibung der Schnittstelle der Queue und der Funktionen `parseTriangle()` und `countPoints()` finden Sie in den angehängten Manpages.

```c
#include <errno.h>
#include <pthread.h>
#include <unistd.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>

#include "sem.h"
#include "triangle.h"
#include "queue.h"

/* Funktionen aus sem.h */
SEM *semCreate(int initVal); // sets errno on failure
void semDestroy(SEM *sem);
void P(SEM *sem);
void V(SEM *sem);

// Funktionen triangle.h (parseTriangle, countPoints): SIEHE MANPAGE
// Funktionen queue.h (qCreate, qPut, qGet, qDestroy): SIEHE MANPAGE

static const size_t MAX_LINE     = 1024;
static const size_t CALC_THREADS = 5;

static void die(const char message[]) {
  perror(message); exit(EXIT_FAILURE);
}

static void usage(void) {
  fprintf(stderr, "Usage: ./parrots <file>\n");  exit(EXIT_FAILURE);
}
```

// Funktions- & Strukturdekl., globale Variablen, etc.

// Hauptfunktion (main)

if(argc != 2) { usage(); }

// Initialisierung

// Threads starten

// Arbeitspakete aus der Datei auslesen

```
  // "Haupt"schleife
```



```
  // Threads + Ressourcen aufräumen
```

```
// Ende Hauptfunktion
```

M:

```
// Funktion Rechenthread
```

```
// Ende Rechenthread
```

R:

```
// Ausgabethread
```

```
// Ende Ausgabethread
```

P:

Schreiben Sie ein Makefile, welches die Targets `all` und `clean` unterstützt. Ebenfalls soll ein Target `parrots` unterstützt werden, welches das Programm `parrots` baut. Greifen Sie dabei stets auf Zwischenprodukte (z.B. `parrots.o`) zurück. Gehen Sie davon aus, dass die vorgegebenen Module `sem.o`, `queue.o` und `triangle.o` stets vorliegen und daher nicht erzeugt werden müssen.

Das Target `clean` soll alle erzeugten Zwischenergebnisse und das Programm `parrots` löschen.

Definieren und nutzen Sie dabei die Variablen `CC` und `CFLAGS` konventionskonform. Achten Sie darauf, dass das Makefile ohne eingebaute Variablen und Regeln (Aufruf von `make -Rr`) funktioniert!

Mk:

Aufgabe 3: Adressräume & Freispeicherverwaltung (23 Punkte)

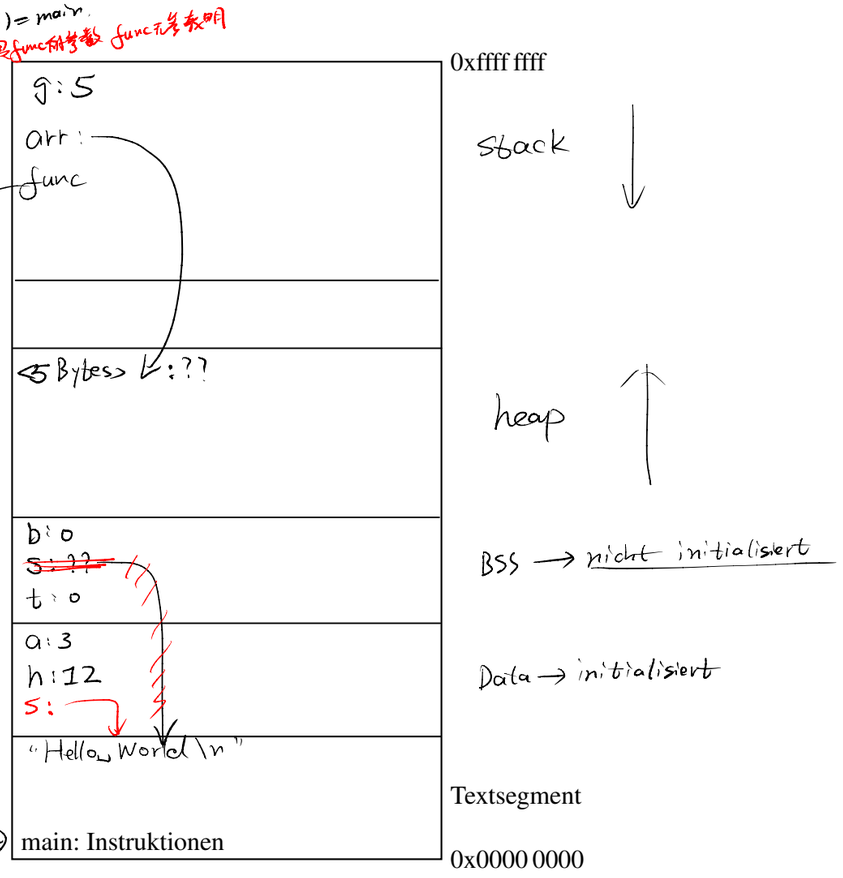
1) Gegeben sei das nachfolgende Programm. Skizzieren Sie den Aufbau des logischen Adressraums eines Prozesses, der dieses Programm ausführt. Tragen Sie die Segmente und deren Namen (analog zum schon vorgegebenen Textsegment) in unten stehende Zeichnung ein. Unterscheiden Sie hierbei die Bereiche zur Speicherung von initialisierten und nicht initialisierten Variablen. Zeichnen Sie für jede Variable ein, wo diese ungefähr im logischen Adressraum zu finden sein wird und welchen Wert sie enthält. Illustrieren Sie im Falle von Zeigervariablen mittels Pfeil, auf welches Datum die Variable jeweils zeigt.

Vermerken Sie zudem, in welche Richtung Segmente variabler Größe wachsen. Gehen Sie hierbei von einem x86-System aus. (8 Punkte)

```
static int a = 3;
static int b = 0;

const char *s = "Hello World\n";
static int t = 0;

int main(void) {
    int g = 5;
    static int h = 12;
    void *arr = malloc(5);
    int (*func)(void) = main;
}
```

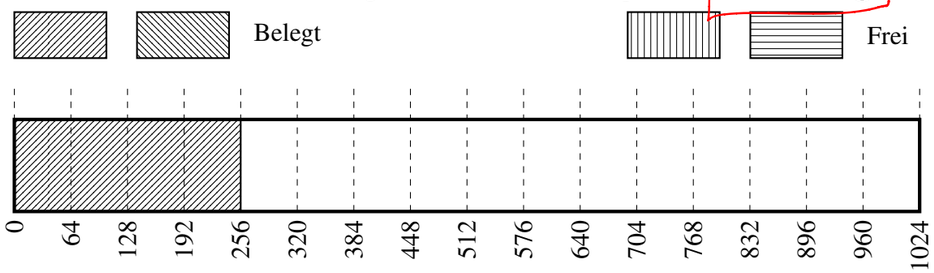


2) Ein in der Praxis häufig eingesetztes Verfahren zur Verwaltung von freiem Speicher ist das *Buddy*-Verfahren.

Nehmen Sie einen Speicher von 1024 Bytes an und gehen Sie davon aus, dass die Freispeicher-Verwaltungsstrukturen separat liegen. Initial ist bereits ein Datenblock der Größe 256 Bytes vergeben worden. Ein Programm führt nacheinander die im folgenden Bild angegebenen Anweisungen aus. (11 Punkte)

⓪ p0 = malloc(200); // 0 (initial vergebener Block)
① p1 = malloc(32);
② p2 = malloc(120);
③ p3 = malloc(60);
④ free(p2);
⑤ p4 = malloc(250);
⑥ free(p1);
⑦ free(p3);

Tragen Sie hinter den obigen Anweisungen jeweils ein, welches Ergebnis die `malloc()`-Aufrufe zurückliefern. Skizzieren Sie in der folgenden Grafik, wie der Speicher **nach Schritt ⑤** aussieht.



Tragen Sie in unten stehender Tabelle die **Adressen** der freien Blöcke (*left-over holes*) nach **jedem** Schritt ein. Für Blöcke gleicher Größe schreiben Sie die Adressen nebeneinander in die Tabellenzeile (es ist nicht notwendig, verkettete Buddys wie in der Vorlesung beschrieben einzutragen).

Größe der Blocks	initial ⓪	①	②	③	④	⑤	⑥	⑦
2^5								
2^6								
2^7								
2^8	256							
2^9	512							
2^{10}								

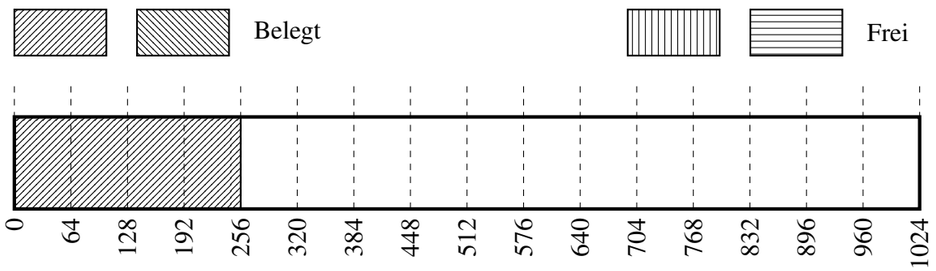
freier Block der Größe 2^9 ab Adresse 512

Hinweis: $2^5 = 32$, $2^6 = 64$, $2^7 = 128$, $2^8 = 256$, $2^9 = 512$, $2^{10} = 1024$

3) Man unterscheidet bei Adressraumkonzepten und bei Zuteilungsverfahren zwischen externer und interner Fragmentierung. Beschreiben Sie beide Arten der Fragmentierung und erklären Sie, ob diese bei der Anwendung des Buddy-Verfahren auftritt. (4 Punkte)

Ersatzgrafik für Teilaufgabe 3.2.

Sie dürfen diese Grafik nutzen, falls Sie sich beim Ausfüllen der Grafik in 3.1. verzeichnet haben. Markieren Sie eindeutig, welche der Grafiken zur Bewertung herangezogen werden soll!



Aufgabe 4: Prozesszustände (7 Punkte)

Beschreiben Sie die Prozesszustände bei der Einplanung von Prozessen sowie die Ereignisse, die jeweils zu Zustandsübergängen führen (Skizze mit kurzer Erläuterung der Zustände und Übergänge).