数据结构课程设计



班级： 1618403

学号： 161840227

姓名： 韦 鑫

指导教师：孙 涵

目录

1.采用的数据结构 ………………………………………… 3

2.算法设计思想 …………………………………………… 3

3.关键代码 ………………………………………………… 4

4.测试数据和结果 ………………………………………… 12

5.算法的时间复杂度及其改进方法 ……………………… 12

6.结束语 …………………………………………………… 13

一、采用的数据结构

双向链表、单链表

//当前进程的双向链表节点结构

typedef struct Dnode

{

int id; //进程id(PID)

char name[70]; //进程名

int conttime; //持续时间，以秒为单位

int memory; //内存大小，以KB为单位

Dnode \*pre, \*next;

}Dnode,\*DList;

//结束进程单链表节点结构

typedef struct node

{

int id; //进程id(PID)

char name[70]; //进程名

int conttime; //持续时间，以秒为单位

int overtime; //结束时间，以秒为单位

node \*next;

}node,\*List;

二、算法设计思想

①当前进程的双向链表按内存从大到小的顺序进行排序，插入新的节点时需要主要链尾的插入

②结束进程的单链表按照结束从最新到旧的顺序排序，所以只需要使用头插法即可。

③两个链表的更新策略：先调用函数获取系统进程信息以及内存信息(有些进程无调取内存的无权限)，并用双向链表进行存储。以该新进程链Cur与上个时间的进程链进行比较Main，他们的交集为还在运行的进程，以Main链中的每个进程A遍历Cur链进行对比，若与A相同则删除Cur中的该进程，若遍历以便Cur未发现A，则说明A进程已结束，在Main链中将A删去，并加入Over结束进程链。遍历所有Main链的进程后，将Cur剩余的进程(即新进程)加入Main链。

三、关键代码

//当前进程的双向链表节点结构

typedef struct Dnode

{

int id; //进程id(PID)

char name[70]; //进程名

int conttime; //持续时间，以秒为单位

int memory; //内存大小，以KB为单位

Dnode \*pre, \*next;

}Dnode,\*DList;

//结束进程单项链表节点结构

typedef struct node

{

int id; //进程id(PID)

char name[70]; //进程名

int conttime; //持续时间，以秒为单位

int overtime; //结束时间，以秒为单位

node \*next;

}node,\*List;

int t = 0; //程序运行的总时间，以秒为单位

int filenum = 0;//文件个数

//显示当前、已结束进程情况

void print(DList CurP, List OverP)

{

printf("当前系统进程：\n");

if (CurP->next != NULL)

{

Dnode \*p = CurP->next;

printf("内存使用\t进程名\t\t\t\t\t\t\t\t\t\t持续时间 PID\n");

printf("==============================================================================================================\n");

while (p)

{

if (p->memory == -1)printf(" 无权限\t");

else printf("%6d K\t", p->memory);

printf("%-52s\t\t\t\t%5d\t %5d\n", p->name, p->conttime,p->id);

p = p->next;

}

}

else printf("当前无进程！\n");

printf("已结束进程：\n");

if (OverP->next != NULL)

{

node \*p;

p = OverP->next;

printf("进程名\t\t\t\t\t\t\t\t结束时间 持续时间 PID\n");

printf("=============================================================================================\n");

while (p)

{

printf("%-52s\t\t%5d s\t%5d s\t\t%5d\n", p->name, p->overtime, p->conttime,p->id);

p = p->next;

}

}

else printf("无结束进程！\n");

}

//将进程信息存入文件

void outlog(DList CurP, List OverP)

{

FILE \*fp;

char filename[] = "a.txt";

filename[0] = 'a' + filenum;

filenum++;

if ((fp = fopen(filename, "w")) == NULL)

{

printf("创建文件失败!\n");

exit(-1);

}

//加入系统时间

time\_t timep;

struct tm \*p;

time(&timep);

p = gmtime(&timep);

fprintf(fp,"%d/%d/%d %d:%d:%d\n", 1900 + p->tm\_year, 1 + p->tm\_mon, p->tm\_mday, 8 + p->tm\_hour, p->tm\_min,p->tm\_sec);

fprintf(fp,"当前系统进程：\n");

if (CurP->next != NULL)

{

Dnode \*p = CurP->next;

fprintf(fp,"内存使用 进程名 持续时间(s) PID\n");

fprintf(fp,"====================================================================================\n");

while (p)

{

if (p->memory == -1) fprintf(fp," 无权限 ");

else fprintf(fp,"%6d K ", p->memory);

fprintf(fp," %-70s %d %d\n", p->name, p->conttime, p->id);

p = p->next;

}

}

else fprintf(fp,"当前无进程！\n");

fprintf(fp,"已结束进程：\n");

if (OverP->next != NULL)

{

node \*p;

p = OverP->next;

fprintf(fp,"进程名 结束时间(s) 持续时间(s) PID\n");

fprintf(fp,"=====================================================================\n");

while (p)

{

fprintf(fp,"%-70s %d\t %d\t %d\n", p->name, p->overtime, p->conttime,p->id);

p = p->next;

}

}

else fprintf(fp,"无结束进程！\n");

fclose(fp);

}

//按照内存从大到小的顺序插入当前进程

void DList\_Insert(DList &L, Dnode \*s)

{

if (L == NULL)

{

printf("error!\n");

exit(0);

}

Dnode \*p = L;

while (p->next != NULL && p->next->memory > s->memory) p = p->next;

if (p->next != NULL) p->next->pre = s;

s->next = p->next;

s->pre = p;

p->next = s;

}

//初始化进程链

bool InitCurP(DList &L)

{

//为当前进程链表头节点分配空间

L = (DList)malloc(sizeof(Dnode));

if (L == NULL)

{

printf("分配内存失败！\n");

exit(-1);

}

L->next = L->pre = NULL;

PROCESSENTRY32 curP; //存放快照进程信息的结构体

curP.dwSize = sizeof(curP); //在使用这个结构之前，先设置它的大小

HANDLE Pshot = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);//获取系统进程快照

HANDLE handle = GetCurrentProcess();//获取当前进程

PROCESS\_MEMORY\_COUNTERS pmc;

if (Pshot == INVALID\_HANDLE\_VALUE)

{

printf("获取快照失败!\n");

exit(0);

}

bool flag = Process32First(Pshot, &curP);//获取第一个进程信息

while (flag)

{

handle = OpenProcess(PROCESS\_ALL\_ACCESS, 0, curP.th32ProcessID);//handle为编号为PID进程的权柄

GetProcessMemoryInfo(handle, &pmc, sizeof(pmc));//获取内存大小

Dnode \*s = (DList)malloc(sizeof(Dnode));

if (s == NULL)

{

printf("分配内存失败！\n");

exit(-1);

}

s->id = curP.th32ProcessID;

string name = curP.szExeFile;

const char \*tname = name.c\_str();

strcpy(s->name, tname);

s->conttime = 0;

s->next = s->pre = NULL;

if (handle)s->memory = pmc.WorkingSetSize / 1024;

else s->memory = -1; //无权限无法提取内存大小

DList\_Insert(L, s);

flag = Process32Next(Pshot, &curP); //遍历下一个进程

}

CloseHandle(Pshot); //清除Pshot句柄

return 1;

}

//更新进程时，在新进程链中，删除旧进程链与新的进程链元素的交集，新进程链中剩下的便是新进程

//并将剩下的插入到旧进程链中

void DList\_Del(DList &L, int id)

{

if (L == NULL)

{

printf("error!\n");

exit(0);

}

Dnode \*p = L,\*q;//p为要删节点的前驱，q为要删除的节点

while (p->next != NULL && p->next->id != id) p = p->next;

q = p->next;

//若存在该节点

if (q != NULL)

{

//若要删除的为链尾

if (q->next == NULL)

{

p->next = NULL;

q->pre = NULL;

free(q);

}

else

{

p->next = q->next;

q->next->pre = p;

free(q);

}

}

}

//更新时，将旧进程链中有、但新进程链没有的进程插入结束进程链

//并将这些进程从旧进程链中删去

void List\_Insert(List &L, node \*s)

{

if (L == NULL)

{

printf("error!\n");

exit(0);

}

node \*p = L;

while (p->next != NULL && p->next->overtime > s->overtime) p = p->next;

s->next = p->next;

p->next = s;

}

//更新当前进程链表mainP和结束进程链表OverP

//将旧进程中的所有进程p逐一与新进程cur对比，若遍历了新进程都未发现该进程(flag = 1)，说明该进程结束,则将其插入结束链表OverP；

//若有相同的，则把当前进程链表的该进程删掉，最后将剩下的进程(这些剩下的为新进程)插入上一秒的进程链表mainP中

void update(DList &mainP, List &OverP)

{

//获取当前正在运行的进程，创建新的进程链

DList curP;

InitCurP(curP);

Dnode \*p = mainP->next;

Dnode \*cur;

bool flag;

while (p)

{

flag = 1;//1表示遍历新进程链均未发现该进程

cur = curP->next;

while (cur)

{

if (p->id == cur->id && strcmp(p->name,cur->name) == 0)//防止新打开的进程和刚关掉的进程分配到同个PID

{

//若新进程链中存在该进程，则将其在旧进程链中的增加持续时间、更新内存使用情况，并在新进程链curP中删掉

flag = 0;

p->conttime += 5;

p->memory = cur->memory;

DList\_Del(curP, p->id);

break;

}

cur = cur->next;

}

if (flag)//该进程需插入结束链

{

node \*s = (List)malloc(sizeof(node));

s->id = p->id;

strcpy(s->name, p->name);

s->conttime = p->conttime + 5; //间隔为5s

s->overtime = t;

s->next = NULL;

List\_Insert(OverP, s);

p = p->next;//将结束的进程从mainP删除前向往下移动一个

DList\_Del(mainP, s->id);

}

else p = p->next;

}

while (curP->next)

{

cur = curP->next;

Dnode \*s = (DList)malloc(sizeof(Dnode));

s->id = cur->id;

strcpy(s->name, cur->name);

s->memory = cur->memory;

s->conttime = cur->conttime;

s->next = s->pre = NULL;

DList\_Insert(mainP, s);

DList\_Del(curP, cur->id);

}

free(curP);

}

int main()

{

DList mainP;//当前进程链

List OverP = (List)malloc(sizeof(node));//结束进程链

if (!InitCurP(mainP))

{

printf("当前进程链表初始化失败！\n");

exit(-1);

}

if (!OverP)

{

printf("结束进程链表初始化失败！\n");

exit(-1);

}

OverP->next = NULL;

outlog(mainP, OverP);

print(mainP, OverP);

while (1)

{

t += 5; //间隔5s执行一次操作

Sleep(5000);//以毫秒为单位

system("cls");

update(mainP, OverP);

outlog(mainP, OverP);

print(mainP, OverP);

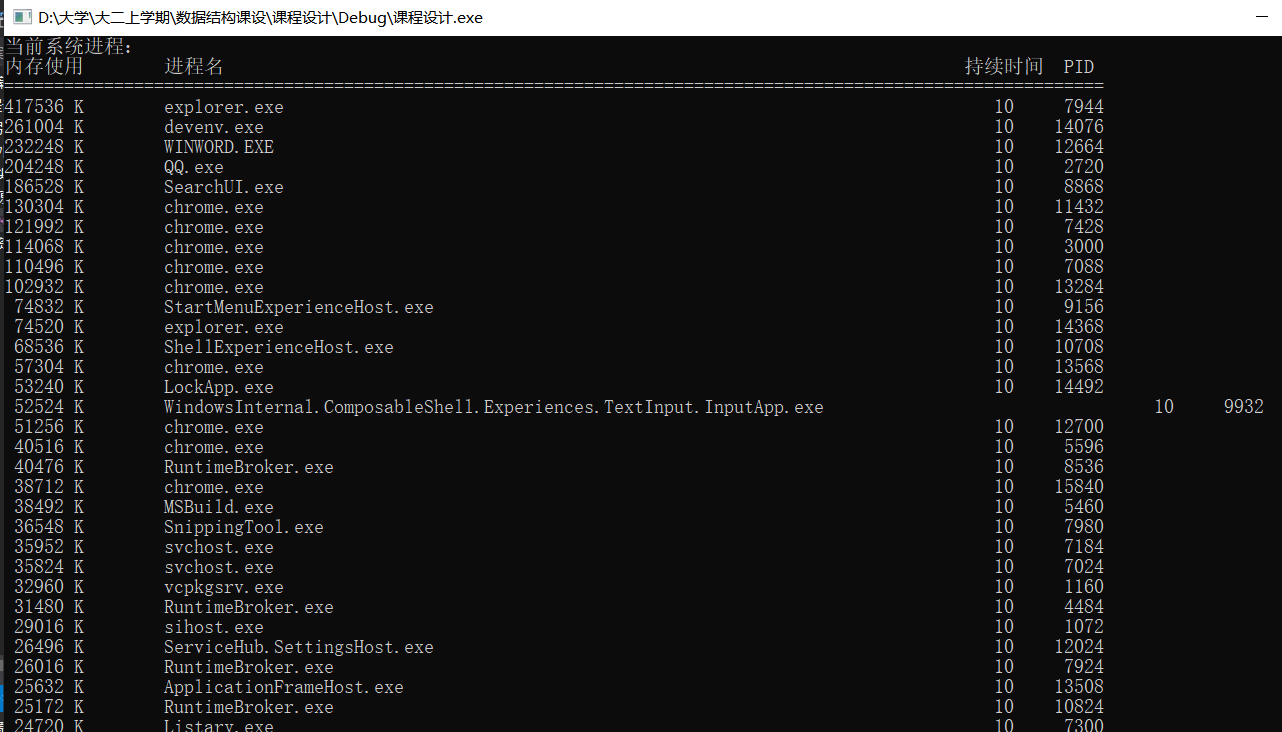
}

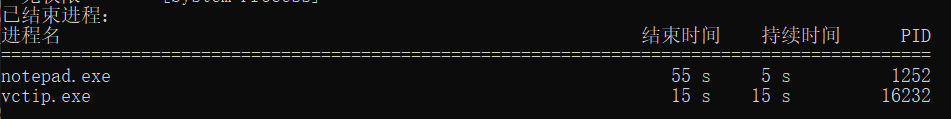
system("pause");

return 0;

}

四、测试数据和结果





五、算法的时间复杂度即改进方法

双向链表插入：O(n)

双向链表删除：O(n)

单链表插入：O(1)

综上，进程更新的时间复杂度为：O(n\*(n+n)) = O(n\*n)

六、结束语

代码共336行

通过该实验，我复习了单链表、双向链表的插入、删除操作，特别在编程时注意了链尾的插入和删除，避免了重蹈覆辙。

另外还搜集资料学习了调用Window API获取进程的内存信息以及使用#include <tlhelp32.h>获取系统进程各信息。

当时想着是否要使用Qt实现与任务管理器一样的程序，但是限于当前的水平能力，最终还是无法实现，所以还需在今后的学习中更加努力！