# Pages

软英1602 陈格平 20165187

Purpose

The goal of the assignment is to design and program a memory management simulation using pages. We need to assign blocks/frames/pages of memory to satisfy each request and ensure no memory locations are concurrently assigned to multiple requests. Once the memory is freed, the location can be used again. By simulate the paging process, we can clearly understand the management of memory using paging method.

Methodology

* I use 2 queue and a vector to store the process. The newProcs stores the process that haven’t enter the memory. The runningProcess stores the process that enter the memory. The exitProcs stores the process that finished.
* When a new process arrive, the front of the newProcs will be pop to the running process. When it arrive, allocate the memory for the new process.
  + Pseudocode

while (simTime == newProcs.front().getArrival()) {

newProcs.front().allocate(pageSize,memory,freespace);

cout<< "for process "<<countprocess<< "the memoryTable is "<<endl;

newProcs.front().printMemoryTable();

runningProcess.push\_back(newProcs.front());//this is the process begin to run

newProcs.pop\_front();

countprocess++;

}

simTime++;

* Allocate function: pass the page size of the memory, memory, and the free space now for the function. When enter in the function, calculate the page that we needed for the memory. If the free space is less than zero, return false. Else return true. After calculation, start the loop from the beginning of the memory table. If the array space of the location is equal to zero, allocate the process here, reduce the needed page, and record the location to the process table. Stop the loop until the pages are all allocated already.
  + Pseudocode

bool process::allocate(int pagesize, int memory[],int &freespace){

int i=0;

int needPage=processSize/pagesize;

int rest=processSize%pagesize;

if(rest!=0)

needPage+=1;//calculate the needed page

if(freespace<0){

return false;

}

freespace-=needPage;

while (needPage!=0){

if(memory[i]==0){

memoryTable.push\_back(i);//memory table

memory[i]=pagesize;//allocate

if(rest!=0){

if(needPage==1){

memory[i]=rest;

fragment=(pagesize-rest);

}

}

needPage--;

}

i++;

}

return true;

}

* Loop the vector of running process. If the process is exit, free the memory of the process that it is located at, and push the exit process at the back of the queue of exit process
  + Pseudocode

for(int i=0;i<runningProcess.size();i++){

if (runningProcess[i].exit()) {

runningProcess[i].setFinished(simTime+1);

exitProcs.push\_back(runningProcess[i]);

runningProcess[i].leave(memory,freespace);

/\* cout<<"leave now!! "<<endl;

printMemory(memory);

}\*/

}

}

* Leave function: pass the memory and the free space of the memory into the function. Get the location the process in from the memory table, set the memory to zero at the location of the process. Calculate the free space of the memory.
  + Pseudocode

void process::leave(int memory[],int &freespace){

//获取指定页

int location;

fragment=0;

for(int i=0;i<memoryTable.size();i++){

location=memoryTable[i];

memory[location]=0;//reset to 0

}

freespace+=memoryTable.size();

memoryTable.clear();

}

Consideration

* How to keep track of memory (free and assigned)?

when a new process enters, set the data at the position of the memory as the page size of the process. Therefore, every time a new process enters, loop the memory, when the data of a position is equal to 0, which means it is free, allocate a part of process here. If the data of the location is not equal to 0, it is assigned.

* What needs to go in “page table”, how many?

Every process owns a unique page table that records the location of the process. So, when we need to free the memory after the process finished, we can get the location of the process from the page table. We can also know the amount of the space being freed by calculating the size of the page table.

* If you run the same simulation data through the various page sizes.
  + How much internal fragmentation? Calculate the free space every time a process allocates and leaves the memory. When a process enters, the free space will reduce by the needed pages, when a process leave, the free space will increase by the size of its page table.
  + How many pages are required? We can calculate the pages needed
  + Did you ever run into a situation where you completely ran out of available space? To test my program, Yes. But it seldom happens in the real life. If the memory is full, it will just wait until the space is enough to enter in. to implement this, I set a flag for every process and reset it to false. loop the running process. If the flag is false, which means it hasn’t been allocate yet, if the process can not be allocated, wait until it is available.

Pseudocode

 while (simTime == newProcs.front().getArrival()) {

        /\*    runningProcess.push\_back(newProcs.front());

            newProcs.pop\_front();

        for(int i=0;i<runningProcess.size();i++){

            if(!runningProcess[i].inRunning()){

                if(runningProcess[i].allocate(pageSize,memory,freespace)){

                     cout<< "for process "<<countprocess<< "the memoryTable is "<<endl;

                    runningProcess[i].printMemoryTable();

                    countprocess++;

                    runningProcess[i].setRunning(true);

                }else{

                    break;

                }

            }

        }\*/

* My class of the process.

class process{

public:

process();//constructor

process(int size,int arrival,int duration);

bool allocate(int pagesize,int memory[],int &freespace);//allocate the page into the memory

void leave(int memory[],int &freespace);

void printMemoryTable();

inline bool exit() {rem--; return finished();}

inline bool finished() {return (rem == 0);}

inline void setFinished(unsigned int finished) {fin = finished;}

inline unsigned int getProcessSize()const {return processSize;}

inline unsigned int getArrival() const {return arr;}//arrival time

inline unsigned int getDuration() const {return dur;}//service time

inline unsigned int getRemaining() const {return rem;}//remaining time

inline unsigned int getFinish() const {return fin;}

inline getFragment(){return fragment;}

private:

unsigned int arr; //arrival

unsigned int dur; //duration/service time

unsigned int rem; //remaining time -- initially duration, finished when 0

unsigned int fin;

unsigned int processSize;

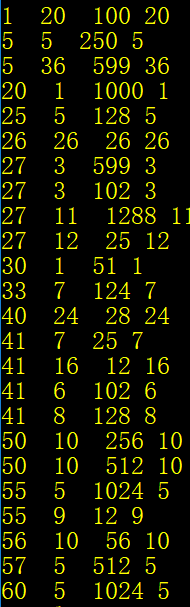
vector <int> memoryTable;

int fragment;

};

void printMemory(int memory[]);

Experiment data & result

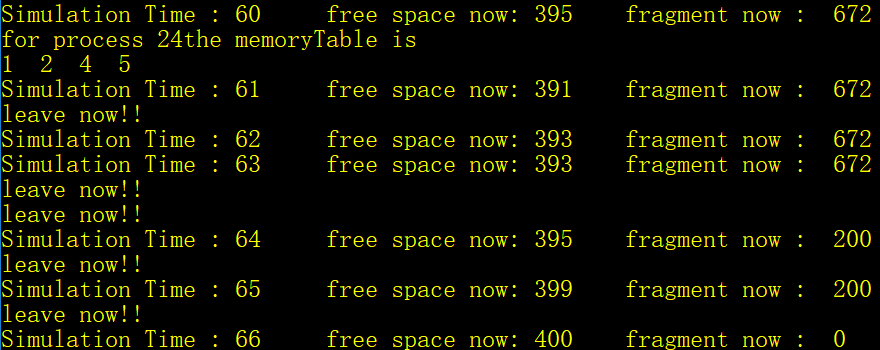
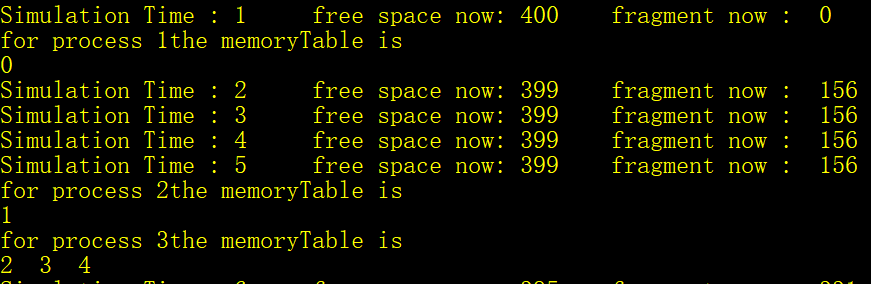
Experiment data: 

Result:

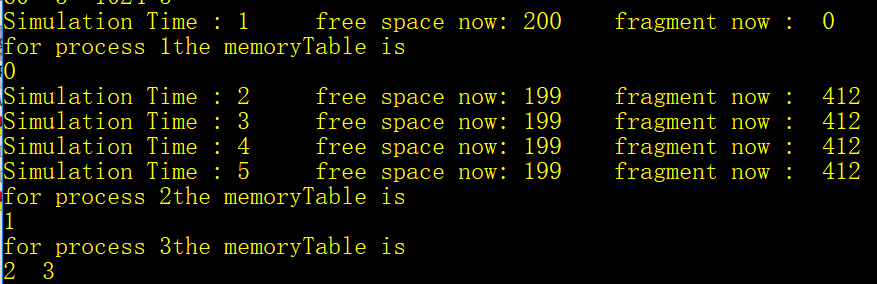
* Page size is 128



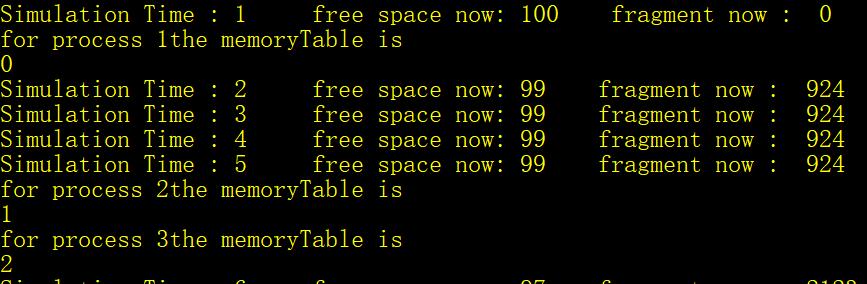
* Page size is 256



* Page size is 512



* Page size is 1024



Conclusion, comments, suggestion

From the simulation of the paging management, we can see that, the larger the size of the page is, the less the pages needed for the process are, also the fragment becomes larger. To summarize, with simple paging, main memory is divided into many small equal-size frames. Each process is divided into frame-size pages. Smaller processes require fewer pages; larger processes require more. When a process is brought in, all its pages are loaded into available frames, and a page table is set up. This approach solves many of the problems inherent in partitioning.