DIGITAL ASSIGNMENT 4

CSE 2005: OPERATING SYSTEMS

ANUSHKA DIXIT [19BCE0577]

Note: Banker’s algorithm has been included in Lab DA 3

Question: Implement the three memory allocation algorithms

First Fit

This method keeps the free/busy list of jobs organized by memory location, low-ordered to high-ordered memory. In this method, first job claims the first available memory with space more than or equal to its size. The operating system doesn't search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size

Code:

```
anushka_os@DESKTOP-96L9A8G: ~
  GNU nano 4.8
#include<stdio.h>

void main()
{
        int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

        for(i = 0; i < 10; i++)
        {
                flags[i] = 0;
                allocation[i] = -1;
        }

        printf("Enter no. of blocks: ");
        scanf("%d", &bno);

        printf("\nEnter size of each block: ");
        for(i = 0; i < bno; i++)
                scanf("%d", &bsize[i]);

        printf("\nEnter no. of processes: ");
        scanf("%d", &pno);

        printf("\nEnter size of each process: ");
        for(i = 0; i < pno; i++)
                scanf("%d", &psize[i]);
        for(i = 0; i < pno; i++)         //allocation as per first fit
                for(j = 0; j < bno; j++)
                        if(flags[j] == 0 && bsize[j] >= psize[i])
                        {
                                allocation[j] = i;
                                flags[j] = 1;
                                break;
                        }

        //display allocation details
        printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");
        for(i = 0; i < bno; i++)
        {
                printf("\n%d\t\t%d\t\t", i+1, bsize[i]);
                if(flags[i] == 1)
                        printf("%d\t\t\t%d",allocation[i]+1,psize[allocation[i]]);
                else
                        printf("Not allocated");
        }
}
```

Output:

```
anushka_os@DESKTOP-96L9A8G: ~
anushka_os@DESKTOP-96L9A8G:~$ nano
anushka_os@DESKTOP-96L9A8G:~$ gcc first.c -o first
anushka_os@DESKTOP-96L9A8G:~$ ./first
Enter no. of blocks: 3

Enter size of each block: 12
7
4

Enter no. of processes: 3

Enter size of each process: 7
4
9

Block no.       size            process no.             size
1               12              1                       7
2               7               2                       4
3               4               Not allocatedanushka_os@DESKTOP-96L9A8G:~$
```

Best fit Algorithm

This method keeps the free/busy list in order by size – smallest to largest. In this method, the operating system first searches the whole of the memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory, making it able to use memory efficiently. Here the jobs are in the order from smallest job to largest job.

Code:

anushka_os@DESKTOP-96L9A8G: ~

GNU nano 4.8

```
#include<stdio.h>

void main()
{
        int fragment[20],b[20],p[20],i,j,nb,np,temp,lowest=9999;
        static int barray[20],parray[20];

        printf("\n\t\t\tMemory Management Scheme - Best Fit");
        printf("\nEnter the number of blocks:");
        scanf("%d",&nb);
        printf("Enter the number of processes:");
        scanf("%d",&np);

        printf("\nEnter the size of the blocks:-\n");
        for(i=1;i<=nb;i++)
    {
                printf("Block no.%d:",i);
        scanf("%d",&b[i]);
    }

        printf("\nEnter the size of the processes :-\n");
        for(i=1;i<=np;i++)
    {
        printf("Process no.%d:",i);
        scanf("%d",&p[i]);
    }

        for(i=1;i<=np;i++)
        {
                for(j=1;j<=nb;j++)
                {
                        if(barray[j]!=1)
                        {
                                temp=b[j]-p[i];
                                if(temp>=0)
                                        if(lowest>temp)
                                        {
                                                parray[i]=j;
                                                lowest=temp;
                                        }
                        }
                }

                fragment[i]=lowest;
                barray[parray[i]]=1;
                lowest=10000;
        }

```

```
        }

        printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
        for(i=1;i<=np && parray[i]!=0;i++)
                printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,p[i],parray[i],b[parray[i]],fragment[i]);
}
```

Output:

```
anushka_os@DESKTOP-96L9A8G:~$ ./best

                    Memory Management Scheme - Best Fit
Enter the number of blocks:5
Enter the number of processes:4

Enter the size of the blocks:-
Block no.1:10
Block no.2:15
Block no.3:5
Block no.4:9
Block no.5:3

Enter the size of the processes :-
Process no.1:1
Process no.2:4
Process no.3:7
Process no.4:12

Process_no      Process_size    Block_no        Block_size      Fragment
1               1               5               3               2
2               4               3               5               1
3               7               4               9               2
4               12              2               15              3anushka_os@DESKTOP-96L9A8G:~$
```

Worst fit

In this allocation technique the process traverse the whole memory and always search for largest hole/partition, and then the process is placed in that hole/partition. It is a slow process because it has to traverse the entire memory to search largest hole.

Code:

```
anushka_os@DESKTOP-96L9A8G: ~
 GNU nano 4.8                                                   worst.c
#include<stdio.h>

int main(){

   int p,m;
   printf("Enter number of processes:");
   scanf("%d",&p);
   printf("Enter number of Memory blocks:");
   scanf("%d",&m);

   int parr[p];
   struct mem{
          int id;
          int size;
   }marr[m];
   int i;
   for(i=0;i<p;i++)
   {
     printf("Enter size of process %d:",i+1);
     scanf("%d",&parr[i]);
   }
   for(i=0;i<m;i++)
   {
     printf("Enter size of memory %d:",i+1);
     scanf("%d",&marr[i].size);
     marr[i].id=i+1;
   }
   int j;
   for(i=0;i<m;i++)
   for(j=i+1;j<m;j++)
   if(marr[i].size<marr[j].size)
   {
    struct mem t=marr[i];
    marr[i]=marr[j];
    marr[j]=t;
   }
   for(i=0;i<p;i++){
       for(j=0;j<m;j++){
         if(marr[j].size>=parr[i]){
              marr[j].size-=parr[i];
              printf("Allocating process %d to memory %d\n Size remaining in it after allocation %d\n\n",i+1,j+1,marr[j].size);
              break;
         }


       }
      if(j==m)
         {printf("Not allocated %d",i);break;}
```

```
anushka_os@DESKTOP-96L9A8G: ~
 GNU nano 4.8                                    worst.c
                marr[j].size-=parr[i];
                printf("Allocating process %d to memory %d\n Size remaining in it after allocation %d\n\n",i+1,j+1,marr[j].size);
                break;
           }


      }
       if(j==m)
          {printf("Not allocated %d",i);break;}
   }

}
```

Output:

```
anushka_os@DESKTOP-96L9A8G:~$ gcc worst.c -o worst
anushka_os@DESKTOP-96L9A8G:~$ ./worst
Enter number of processes:4
Enter number of Memory blocks:5
Enter size of process 1:1
Enter size of process 2:4
Enter size of process 3:7
Enter size of process 4:12
Enter size of memory 1:10
Enter size of memory 2:15
Enter size of memory 3:5
Enter size of memory 4:9
Enter size of memory 5:3
Allocating process 1 to memory 1
 Size remaining in it after allocation 14

Allocating process 2 to memory 1
 Size remaining in it after allocation 10

Allocating process 3 to memory 1
 Size remaining in it after allocation 3

Not allocated 3anushka_os@DESKTOP-96L9A8G:~$
```

Question 2: Run an experiment to determine the context switch time from one process to another and one kernel thread to another. Compare the findings.

PART 1 – PROCESS CONTEXT SWITCH TIME

Code:

```
anushka--os@LAPTOP-6G5U0QLQ: ~
  GNU nano 4.8
#define _GNU_SOURCE
#include <assert.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

typedef struct {
    unsigned long size,resident,share,text,lib,data,dt;
} ProcStatm;

void ProcStat_init(ProcStatm *result) {
    const char* statm_path = "/proc/self/statm";
    FILE *f = fopen(statm_path, "r");
    if(!f) {
        perror(statm_path);
        abort();
    }
    if(7 != fscanf(
        f,
        "%lu %lu %lu %lu %lu %lu %lu",
        &(result->size),
        &(result->resident),
        &(result->share),
        &(result->text),
        &(result->lib),
        &(result->data),
        &(result->dt)
    )) {
        perror(statm_path);
        abort();
    }
    fclose(f);
}
```

```c
int main(int argc, char **argv) {
    ProcStatm proc_statm;
    char *base, *p;
    char system_cmd[1024];
    long page_size;
    size_t i, nbytes, print_interval, bytes_since_last_print;
    int snprintf_return;

    /* Decide how many ints to allocate. */
    if (argc < 2) {
        nbytes = 0x10000;
    } else {
        nbytes = strtoull(argv[1], NULL, 0);
    }
    if (argc < 3) {
        print_interval = 0x1000;
    } else {
        print_interval = strtoull(argv[2], NULL, 0);
    }
    page_size = sysconf(_SC_PAGESIZE);

    /* Allocate the memory. */
    base = mmap(
        NULL,
        nbytes,
        PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS,
        -1,
        0
    );
    if (base == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
/* Write to all the allocated pages. */
i = 0;
p = base;
bytes_since_last_print = 0;
    snprintf_return = snprintf(
    system_cmd,
    sizeof(system_cmd),
    "ps -o pid,vsz,rss | awk '{if (NR == 1 || $1 == \"%ju\") print}'",
    (uintmax_t)getpid()
);
assert(snprintf_return >= 0);
assert((size_t)snprintf_return < sizeof(system_cmd));
bytes_since_last_print = print_interval;
do {
    /* Modify a byte in the page. */
    *p = i;
    p += page_size;
    bytes_since_last_print += page_size;

    if (bytes_since_last_print > print_interval) {
        bytes_since_last_print -= print_interval;
        printf("extra_memory_committed %lu KiB\n", (i * page_size) / 1024);
        ProcStat_init(&proc_statm);
        /* Check /proc/self/statm */
        printf(
            "/proc/self/statm size resident %lu %lu KiB\n",
            (proc_statm.size * page_size) / 1024,
            (proc_statm.resident * page_size) / 1024
        );
        /* Check ps. */
        puts(system_cmd);
        system(system_cmd);
        puts("");
    }
    i++;
```

```
anushka--os@LAPTOP-6G5U0QLQ: ~
  GNU nano 4.8
            }
            i++;
    } while (p < base + nbytes);

    /* Cleanup. */
    munmap(base, nbytes);
    return EXIT_SUCCESS;
}
```

Output:

```
Ticks elapsed: 18446744073709540212 (5.12267e+15 us)
Ticks elapsed: 18446744073709546476 (5.12267e+15 us)
Ticks elapsed: 18446744073709546966 (5.12267e+15 us)
Ticks elapsed: 18446744073709542092 (5.12267e+15 us)
Ticks elapsed: 18446744073709542510 (5.12267e+15 us)
Ticks elapsed: 18446744073709540814 (5.12267e+15 us)
Ticks elapsed: 18446744073709546692 (5.12267e+15 us)
Ticks elapsed: 18446744073709545608 (5.12267e+15 us)
Ticks elapsed: 18446744073709544454 (5.12267e+15 us)
Ticks elapsed: 18446744073709541190 (5.12267e+15 us)
Ticks elapsed: 18446744073709546296 (5.12267e+15 us)
Ticks elapsed: 18446744073709546424 (5.12267e+15 us)
Ticks elapsed: 18446744073709546332 (5.12267e+15 us)
Ticks elapsed: 2692 (0.74757 us)
Ticks elapsed: 18446744073709536263 (5.12267e+15 us)
Ticks elapsed: 18446744073709540797 (5.12267e+15 us)
Ticks elapsed: 18446744073709542387 (5.12267e+15 us)
Ticks elapsed: 151 (5.12267e+15 us)
Ticks elapsed: 18446744073709538593 (5.12267e+15 us)
Ticks elapsed: 18446744073709540819 (5.12267e+15 us)
Ticks elapsed: 18446744073709543959 (5.12267e+15 us)
Ticks elapsed: 2111 (0.586226 us)
```

PART 2 – THREAD CONTEXT SWITCH TIME

Code:

```
anushka--os@LAPTOP-6G5U0QLQ: ~
  GNU nano 4.8
#include <iostream>
#include <vector>
#include <pthread.h>
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>

double CpuFrequency=3601.0; // CPU frequency in MHz

pthread_cond_t cv;
pthread_mutex_t m;
pthread_t thread2;

struct timeval before, after;

typedef unsigned long long ticks;
unsigned long long beforeTicks, afterTicks;

static __inline__ ticks getrdtsc()
{
     unsigned a, d;
//     asm("cpuid"); // We don't need to cause a pipeline stall for this test
     asm volatile("rdtsc" : "=a" (a), "=d" (d));

     return (((ticks)a) | (((ticks)d) << 32));
}

void *beginthread2(void *v)
{
   for (;;)
   {
      // Wait for a signal from thread 1
      pthread_mutex_lock(&m);
      pthread_cond_wait(&cv, &m);

      // Some dequeue op would normally be performed here after a spurious wake
      // up test
```

```
    // Get the ending ticks
    afterTicks=getrdtsc();
    pthread_mutex_unlock(&m);

    // Display the time elapsed
    std::cout << "Ticks elapsed: " << afterTicks-beforeTicks << " ("
              << (afterTicks-beforeTicks)/CpuFrequency << " us)\n";
  }

  return NULL;

nt main(int argc, char *argv[])

  int core1=0, core2=0;

  if (argc < 3)
  {
    std::cout << "Usage: " << argv[0] << " producer_corenum consumer_corenum" << std::endl;
    return 1;
  }

  // Get core numbers on which to perform the test
  core1 = atoi(argv[1]);
  core2 = atoi(argv[2]);

  std::cout << "Core 1: " << core1 << std::endl;
  std::cout << "Core 2: " << core2 << std::endl;

  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&cv, NULL);

  cpu_set_t cpuset;

  CPU_ZERO(&cpuset);
  CPU_SET(core1, &cpuset);

  // Set affinity of the first (current) thread to core1
  pthread_t self=pthread_self();
  if (pthread_setaffinity_np(self, sizeof(cpu_set_t), &cpuset)!=0)
  {
     perror("pthread_setaffinity_np");
     return 1;
  }

  CPU_ZERO(&cpuset);
  CPU_SET(core2, &cpuset);

  // Create second thread
  pthread_create(&thread2, NULL, beginthread2, NULL);
  // Set affinity of the second thread to core2
  if (pthread_setaffinity_np(thread2, sizeof(cpu_set_t), &cpuset)!=0)
  {
     perror("pthread_setaffinity_np");
     return 1;
  }

  // Run the test
  for (;;)
  {
     // Sleep for one second
     sleep(1);
     // Get the starting ticks
     beforeTicks=getrdtsc();

     // Signal thread 2
     pthread_mutex_lock(&m);
     // Some enqueue op would normally be performed here
     pthread_cond_signal(&cv);
```

```
anushka--os@LAPTOP-6G5U0QLQ: ~
 GNU nano 4.8
        // Some enqueue op would normally be performed here
        pthread_cond_signal(&cv);
        pthread_mutex_unlock(&m);
    }
}
```

Output:

```
Ticks elapsed: 18446744073709545602 (5.12267e+15 us)
Ticks elapsed: 86 (0.0238823 us)
Ticks elapsed: 18446744073709542506 (5.12267e+15 us)
Ticks elapsed: 18446744073709543780 (5.12267e+15 us)
Ticks elapsed: 18446744073709539903 (5.12267e+15 us)
Ticks elapsed: 18446744073709543494 (5.12267e+15 us)
Ticks elapsed: 322 (5.12267e+15 us)
Ticks elapsed: 18446744073709538254 (5.12267e+15 us)
Ticks elapsed: 18446744073709541216 (5.12267e+15 us)
Ticks elapsed: 18446744073709547372 (5.12267e+15 us)
Ticks elapsed: 18446744073709541924 (5.12267e+15 us)
Ticks elapsed: 18446744073709546782 (5.12267e+15 us)
Ticks elapsed: 18446744073709539960 (5.12267e+15 us)
Ticks elapsed: 18446744073709546062 (5.12267e+15 us)
Ticks elapsed: 18446744073709541336 (5.12267e+15 us)
Ticks elapsed: 18446744073709545906 (5.12267e+15 us)
Ticks elapsed: 18446744073709540568 (5.12267e+15 us)
Ticks elapsed: 18446744073709543884 (5.12267e+15 us)
Ticks elapsed: 18446744073709540848 (5.12267e+15 us)
Ticks elapsed: 18446744073709536152 (5.12267e+15 us)
Ticks elapsed: 18446744073709546208 (5.12267e+15 us)
Ticks elapsed: 18446744073709544760 (5.12267e+15 us)
Ticks elapsed: 18446744073709539776 (5.12267e+15 us)
Ticks elapsed: 18446744073709546506 (5.12267e+15 us)
Ticks elapsed: 18446744073709546414 (5.12267e+15 us)
Ticks elapsed: 18446744073709546486 (5.12267e+15 us)
Ticks elapsed: 18446744073709540884 (5.12267e+15 us)
Ticks elapsed: 1900 (0.527631 us)
```

Comparison: Thread Context switch is faster than the Process Context switch.

Question 3: Calculate the hit and miss ratio of cache memory using C

Code:

```
anushka--os@LAPTOP-6G5U0QLQ: ~
  GNU nano 4.8
#include <stdio.h>
#include <stdlib.h>

#define ELEMENTS 10000

int main()
{
  int* arr = calloc(ELEMENTS, sizeof(int));
  int i;
  int sum = 0;
  int iterations = 0;

  for (i = 0; i < ELEMENTS; i++)
    arr[i] = 5;

  for (i = 0; i < ELEMENTS; i++) {
    sum += arr[i];
    iterations++;
  }

  printf("Sum: %d; Iterations: %d\n", sum, iterations);

  free(arr);
  return 0;
}
```

Output:

```
anushka--os@LAPTOP-6G5U0QLQ:~$ gcc hit.c -o hit
anushka--os@LAPTOP-6G5U0QLQ:~$ ./hit
Sum: 50000; Iterations: 10000

              3,603      cache-references:u
              1,056      cache-misses:u

        0.000519530 seconds time elapsed
```

From this we can identify that 29.309 % of all cache refs are misses. From the formula, 1-miss ratio, hit ratio can be calculated. In this case it is 1 – 0.29309 = 0.70691.