MODEL INSTITUTE OF ENGINEERING AND TECHNOLOGY, KOT BHALWAL, JAMMU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

A PROJECT SYNOPSIS ON

STIMULATE HOW PARENT AND CHILD PROCESSES YSE SHARED MEMORY AND ADDRESS SPACE

SUBMITTED TO:

PROF.SAURAB SHARMA

SUBMITTED BY:

KHUSHBOO SHALI (2022A1L009)

LOVISH SHARMA(2021A1R151)

AYUSH BHAT (2021A1R178)

PARAMVIR SINGH(2021A1R173)

ACKNOWLEDGEMENT

Through this Section of our Report ,we want to present our gratitude towards our institution MODEL INSTITITUTE OF ENGINERRING AND TECHNOLOGY (MIET).Being able to work on projects, that require you to analyze and solve a problem that exists in real world, is an experience that one can achieve rarely, and we are happy and thankful to get that chance here.

We would also like to thank our mentors Prof.Ankur Gupta (Principal), Asst.Prof.Saurab Sharma for being a guiding force throughout this period. It was a great experience to work under this project as we learned a lot of new things. We would also like to thank Mr.Ashok Kumar (Dean Academics) and the CSE Department who encouraged and inspired us for learning new skills

STIMULATE HOW PARENT AND CHILD PROCESSES YSE SHARED MEMORY AND ADDRESS SPACE

Khushboo Shali, Ayush Bhat, Lovish sharma,Paramveer
Department of CSE
Model Institute of Engineering & Technology, Kotbhalwal, Jammu, J&K, India-180013

I. Introduction

Simulation tools are widely used to measure and analyze performance characteristics of systems in order to improve overall performance and explore potential performance bottlenecks. Although, existing research reports that system-level mechanisms, such as task switches and process migration, may dominate the performance of modern systems, most simulation tools do not provide good enough support to simulate such events and to analysis their impact.

The fast development of parallel and distributed systems is mainly motivated by the good price/performance ratio they introduce in comparison to other equivalent architectures. Recently small scale shared memory systems were widely used for different types of applications, and modern operating systems such as WindowsNT TM and SCO-Unix TM , support these hardware architectures. Massive parallel machines such as SP2 replace the use of expensive supercomputer machines and recently, different prototypes of distributed shared memory systems, such as FLASH and DASH ([Kusk94], [Leno92]), have been proposed. Unfortunately, the achievable performance of these systems is highly sensitive to different parameters, e.g. number of processors (scalability), communication patterns, sharing characteristics, cache coherency protocol, etc. The capability to simulate such system-level mechanisms and to evaluate their impact on the overall performance of the system is essential to the developing process of modern parallel and distributed systems.

This report introduces a new simulation environment, termed "SMART[1]" that extends existing simulators, such as MINT, with the capability to emulate the effect of system level events and mechanisms. "Smart" provides a user friendly graphical interface (GUI) that provides hooks to control different system-level parameters and mechanisms, such as cache coherency protocols, cache organization, scheduling policies of processes and threads, etc. The Smart environment can be used either for monitoring, analyzing and measuring different system events, or as a powerful visual-based debugging tool. This report describes the "Smart" environment and demonstrates the importance of simulating system-level mechanisms and events to the understanding of the performance factors of modern architectures. We will present the tool, show how it can be

used and we provide several important examples demonstrating the unique features of the simulator. We hope that these examples will convince the reader of the

importance of simulating system-level effects as part of the architecture simulation. In addition, we believe that incorporating system activities within the simulation environment is essential for understanding the performance characteristics and bottlenecks of any computer-based system, and in particular parallel shared memory systems. In this paper we use the Smart to study shared-memory architectures, but its use may be extended to other architectures and software environments.

Most of the existing shared memory simulators are based on traces of memory references, which is a very powerful tool for examining the performance of a single processor cache-based systems. A trace contains a list of memory references made by the processor independently of the cache configuration. Thus, the same trace can be used to compare the impact of the computer architecture on the overall performance. However, traces are not the ultimate mean of examining the performance characteristics of distributed and parallel systems and of understanding the impact of system-level mechanisms on the overall system performance. In such architectures (1) the configuration of the system and different software based mechanisms may significantly affect the memory references the system made, and (2) traces of parallel execution usually require enormous storage space, making this technique too expensive.

In order to overcome these limitations, it was proposed to use a feedback based mechanism that informs the trace generator engine about the state of the machine so that it can change the traces being created on the fly.

A program-driven simulation suggests integrating the simulation and the events tracing into a single tool environment and performing the two activities simultaneously. The simulation is divided into two primary components: The front-end layer that acts as an event generator engine and the back-end layer that can manipulate these events to reflect the change in the state of the machine according to system-level events and mechanisms. Since the front-end and the back-end execute simultaneously, the back-end layer can efficiently monitor the front-end activity and send the appropriate feedback to

[1] http://lasdpc.icmc.usp.br/~ssc640/pos/fork-wait-exec-simulator/

the front-end engine. The correlation between the backend and the front-end is one of the major principles that allows us to accurately measure and simulate the system activities effect on the performance of parallel machines.

Various shared memory simulation tools have been developed during the recent years. For example, The Tango simulator ([Davi91]) aims to simulate shared memory systems. Each simulated processor is simulated at the host computer by an individual process. Each time only a single process is run while the rest remain blocked. As well as the overhead associated with the synchronization of the processes on the local host computer, the researcher cannot control or examine the impact of various system activities, such as processes or threads scheduling policy and management. Another shared-memory simulator called Proteus ([Brew91]) employs an execution-driven simulation technique. Within this technique the original code of the parallel program is modified and preprocessed since special calls to the backend simulator need to be inserted. As Tango, Proteus provides the researcher with the means to simulate the execution of shared memory parallel programs, however neither can they incorporate the effect of system activities, such as process scheduling management, nor can they provide the user capability to control and configure these task scheduling system activities. The MINT simulator is a program driven shared memory simulator ([Veen93]) that can execute MIPS R3000 parallel programs and generate memory events to feed a back-end simulator. We based our Smart simulation environment on the MINT simulator, and special attention is given to that simulator in section 2. Section 3 presents the methodology of Smart, Sections 4,5 and 6 describe different ways the Smart simulator can be used and emphasize the kind of capabilities the Smart can provide, Section 7 concludes this paper.

II. Relate Work

MINT front-end simulator The MINT simulator was developed by J. E. Veenstra and R. J. Fowler at the University of Rochester [Veen93]. It provides an easy way to construct an event driven multiprocessor simulator. MINT can run native executables of MIPS R3000 (usually of IRIX operating systems) by using a hybrid technique that employs both native execution and software interpretation. On top of the MINT simulator the user can develop his own simulator (back-end simulator). The back-end simulator and MINT are linked to create a single executable code of the simulator. MINT's simulator engine provides the back-end multiple streams of memory events corresponding to each of the running threads or processes. The back-end simulator collects the events, and then has the possibility of controlling them and even scheduling its own events.

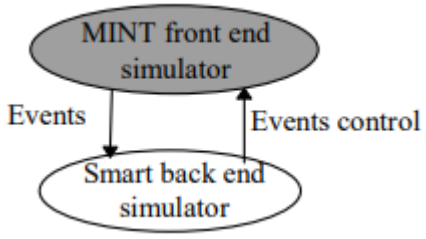


Figure 1: Front-end and Back-end Interface

MINT can simulate both processes and threads. When the running application creates a new thread (using the IRIX sproc()), both the thread parent and the thread child continue to run in the same address space. On the other hand, when a new process is created (for example by calling UNIX fork() system call) the entire virtual address space of the parent process is copied to its child, however these spaces are separate (mapped to different physical locations). Since all these simulated processes lay in the same simulator address space, then when a parent process forks a child, the parent's information should be copied to a new location and to avoid contentions with its child, the child must use different addresses to access its copy. MINT uses an efficient method to overcome this problem by dynamically translating these addresses rather then relocating data at fork time. The benefits of this method are discussed in ([Veen93]).

The interface between the front-end and the back-end is made by calling a user-predefined set of functions. As MINT calls one of these functions it passes a pointer to the information associated with the corresponding event, e.g. event time, the process id of the process or thread that has generated the event, address of data item to be read or written. The programmer of the back-end can assign his own code to the functions called by MINT's events, and in a such way the back-end can be informed and act accordingly to the system's events. In addition, the backend can send feedback to MINT through the returned value of the events' function, by either allowing the thread or process that has generated the event to proceed, or to stall. In the case that the back-end decides to stall the execution of a certain thread (as a result of cache miss for instance), it may also resume the execution of the thread at any other time. MINT allows the back-end simulator to set its own events and schedule them as well. This feature can be very useful, when the back-end simulator wishes to activate its own periodic activities such as threads scheduling, sampling and measuring system performance, etc.

Methodology of the Smart simulator

The system-level events are simulated by the frontend layer (Smart) which was built on top of an existing simulator such as MINT. The MINT serves as an event generator engine that drives Smart with system memory events such as memory references to data, synchronization activities, etc. Smart is capable of acting in accordance with these events and can send feedback to MINT as to whether the events generated by the threads are either allowed to proceed or are rescheduled. Our primary goals in the development of Smart are:

1. To include several significant system related activities, such as task migration management and processes scheduling control, in the simulation environment. Smart can control the scheduling policies of threads or processes. Moreover, Smart provides state of the art scheduling primitives so the user can easily build on top of it its own task scheduler.
2. To provide a convenient and configurable tool for both debugging and monitoring memory system events of shared memory systems.

3. To supply wide range measurements of system-level and single processor-level performance, such as system utilization, cache performance and some other communication parameters.
4. To visualize the sharing pattern of multi-cache systems according to their coherency state and the process they belong to (to be discussed in details in a further discussion in this paper).
5. To provide the user with an easy way to write and add new cache coherency protocol to the "Smart" simulator environment and to debug it.

Environment simulation model

The examples presented in this paper are taken from a shared bus multi-cache system. Our primary intention in these examples is to study the impact of system-level mechanisms on the design parameters of the shared memory. The execution model we choose for these examples assumes a set of identical processing elements, each owning a private cache and a shared global memory. The processors and the main memory are connected through a shared bus, where only a single bus master is allowed to control the bus each time.

The software environment supports simulation of multiple processes or multiple threads. Each processor owns a local threads queue where it holds the threads that were assigned for execution. The thread-scheduler divides the threads of the examined benchmark equally among the processors. Each thread is executed for a fixed time slice (unless it is blocked), and as it is suspended, the scheduler may decide whether to return the thread to the current local queue or migrate the thread to another processor (chosen arbitrarily). All coherency activities are maintained through the shared bus according to the cache coherency protocol. Each running active thread generates several relevant events such as memory references events and synchronization events.

The structure of the Smart front-end layer

As illustrated in figure 3.1, Smart consists of several modules, some written in C and some in C++. These modules are independent of each other and they provide the user with a convenient way to extend or modify each module almost individually. All these modules are linked together to MINT front-end object that drives them with the system events. Some of the main advantages of Smart starts with its module of run-time system configuration management. The system parameters are set in a separate configuration file which is read by the simulator as it is invoked. When the user wishes to examine various system configurations, it does not have to recompile the simulator each time with different parameters, but can easily set the system parameters in the configuration file.

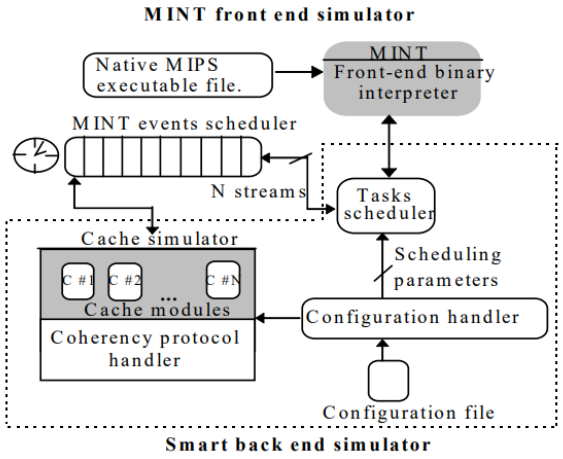


Figure-2: General Structure of Simulator

Proposed Solution:

Proposed Algorithm:

Algorithm-1: Sync of Parant-Child Process using IPC

Input: P_ID, P_ID_ADD
Output: P_ID_time, P_ID_Status

Procedure:

Signal (&Status**)**{
If(P_ID is not NULL)
Index all the P:{P_0 U P_1UP_2…U P_n}
Initialize P<- P_0
Set P_ID is private
Create IPC<- P_0
If (P_ID < 0)
Return (error)
If (P_ID_ADD == -ve)
Return (error)
Else
Call P:{ P_0 U P_1UP_2…U P_n}, and initialize with a private P_ID
Create **IPC**<-{ P_0 U P_1UP_2…U P_n}
End if-else
Initialize P_ID <-**fork();**
If(P_ID<0)
Return(fork_error)
End if
If(P_ID == 0)
Create **IPC**<-fork({P_0 U P_1UP_2…U P_n})
End if
}
Wait(&Status**){**
P_ID_time <- **fork(**IPC<-{ P_0 U P_1UP_2…U P_n}**)**
P_ID_Staus<-**fork(**IPC<-{ P_0 U P_1UP_2…U P_n}**)**
}
Return P_ID_Status, P_ID_time

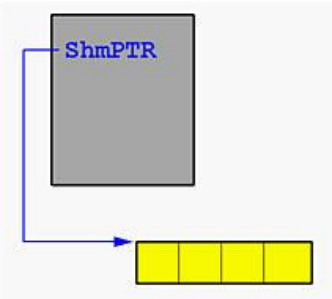
Practical Implementation:

The following main function runs as a server. It uses **IPC_PRIVATE** to request a private shared memory. Since the client is the server's child process created *after* the shared memory has been created and attached, the child client process will receive the shared

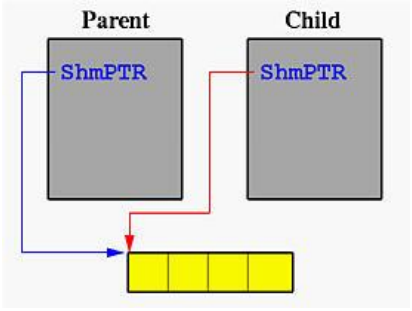
memory in its address space and as a result no shared memory operations are required.

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>

void  ClientProcess(int []);

void  main(int  argc, char *argv[])
{
  int    ShmID;
  int    *ShmPTR;
  pid_t  pid;
  int    status;

  if (argc != 5) {
printf("Use: %s #1 #2 #3 #4\n", argv[0]);
exit(1);
}

ShmID = shmget(IPC_PRIVATE, 4*sizeof(int),
IPC_CREAT | 0666);
if (ShmID < 0) {
printf("*** shmget error (server) ***\n");
exit(1);
}
printf("Server has received a shared memory of four
integers...\n");

ShmPTR = (int *) shmat(ShmID, NULL, 0);
if ((int) ShmPTR == -1) {
printf("*** shmat error (server) ***\n");
exit(1);
}
printf("Server has attached the shared memory...\n");

ShmPTR[0] = atoi(argv[1]);
ShmPTR[1] = atoi(argv[2]);
ShmPTR[2] = atoi(argv[3]);
ShmPTR[3] = atoi(argv[4]);
printf("Server has filled %d %d %d %d in shared
memory...\n",
ShmPTR[0], ShmPTR[1], ShmPTR[2],
ShmPTR[3]);

printf("Server is about to fork a child process...\n");
pid = fork();
if (pid < 0) {
printf("*** fork error (server) ***\n");
exit(1);
}
else if (pid == 0) {
ClientProcess(ShmPTR);
exit(0);
}

wait(&status);
printf("Server has detected the completion of its
child...\n");
shmdt((void *) ShmPTR);
printf("Server has detached its shared memory...\n");
shmctl(ShmID, IPC_RMID, NULL);
printf("Server has removed its shared memory...\n");
printf("Server exits...\n");
exit(0);
}

void  ClientProcess(int  SharedMem[])
{
  printf("  Client process started\n");
printf("  Client found %d %d %d %d in shared
memory\n",
  SharedMem[0], SharedMem[1], SharedMem[2],
SharedMem[3]);
printf("  Client is about to exit\n");
}
```

This program asks for a shared memory of four integers and attaches this shared memory segment to its address space. Pointer **ShmPTR** points to the shared memory segment. After this is done, we have the following:

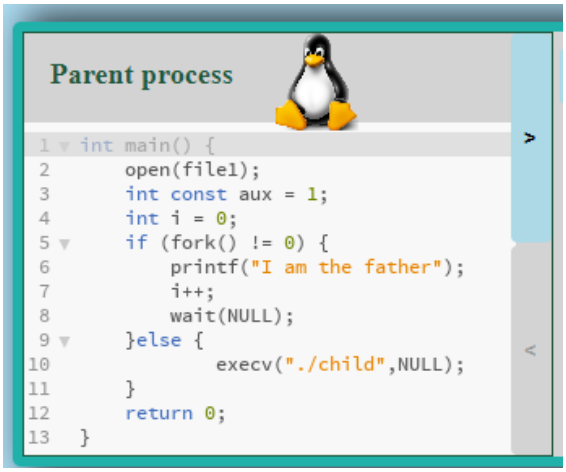


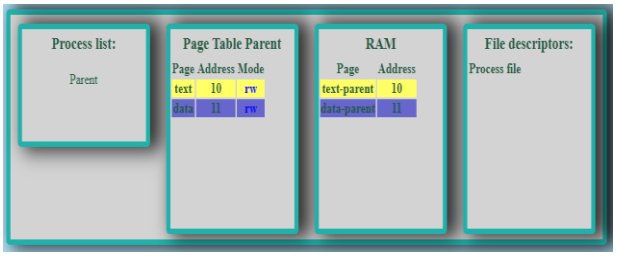
Then, this program forks a child process to run function **ClientProcess()**. Thus, two *identical* copies of address spaces are created, each of which has a variable **ShmPTR** whose value is a pointer to the shared memory. As a result, the child process has already known the location of the shared memory segment and does not have to use **shmget()** and **shmat()**. This is shown below:

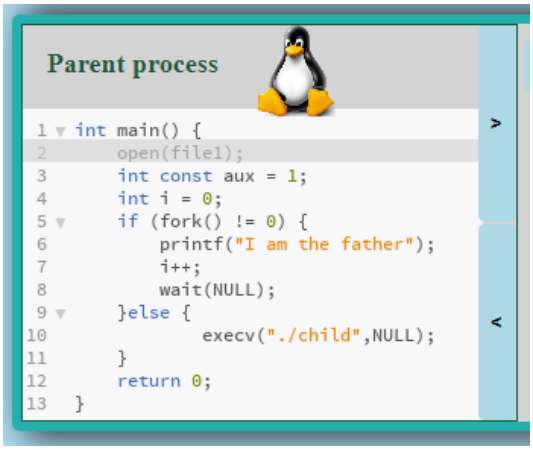


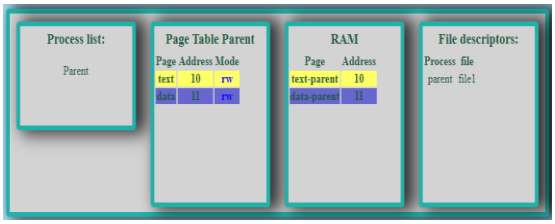
The parent waits for the completion of the child. For the child, it just retrieves the four integers, which were stored there by the parent *before* forking the child, prints them and exits. The **wait()** system call in the parent will detect this. Finally, the parent exits.

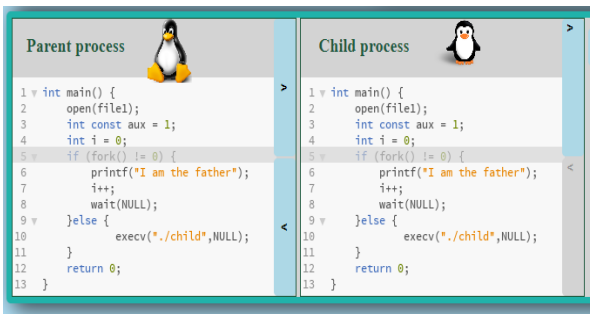
Result and Discussion:

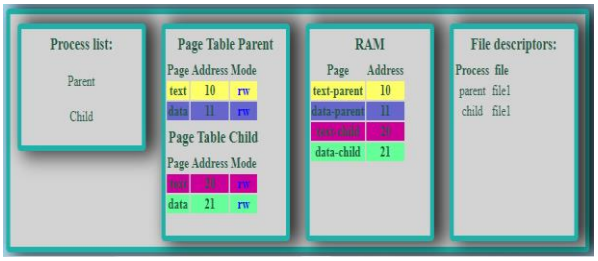


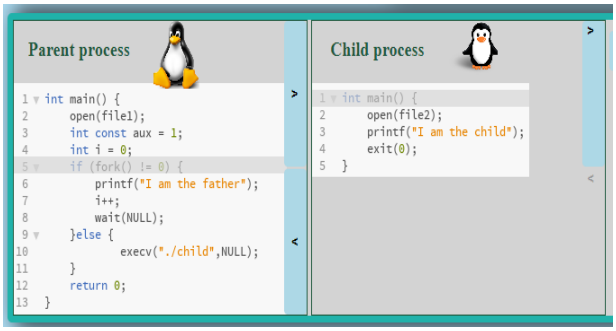


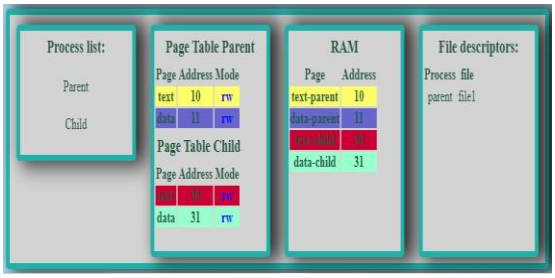












References:

[1] A. N. Kubankov and S. V. Kozlov, "Theoretical Aspects of Process Synchronization in Ensuring the Interoperability of Integrated Control Systems," *2021 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHROINFO*, 2021, pp. 1-6, doi: 10.1109/SYNCHROINFO51390.2021.9488382.

[2] R. Lienhart, I. Kozintsev, S. Wehr and M. Yeung, "On the importance of exact synchronization for distributed audio signal processing," *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).*, 2003, pp. IV-840, doi: 10.1109/ICASSP.2003.1202774.

[3] R. Lienhart, I. Kozintsev, M. Yeung and S. Wehr, "On the importance of exact synchronization for distributed audio signal processing," *2003 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (IEEE Cat. No.03TH8684)*, 2003, pp. 147-, doi: 10.1109/ASPAA.2003.1285842.

[4] A. R. Safin, N. N. Udalov and M. V. Kapranov, "Features of the mutual synchronization of nanooscillators," 2017 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SINKHROINFO), 2017, pp. 1-3, doi: 10.1109/SINKHROINFO.2017.7997555.

[5] A. N. Kubankov and S. V. Kozlov, "Innovative Ways to Ensure the Interoperability of a Complete Group of Processes in the Lifecycle of an Integrated Management System Based on their Synchronization," 2020 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHROINFO), 2020, pp. 1-5, doi: 10.1109/SYNCHROINFO49631.2020.9166045.

[6] A.A. Kamenshchikov, A. Ya. Oleynikov and T.D. Shirobokova, "Standards of interoperability in a high-performance environment", *Software systems: theory and applications*, vol. 9, no. 4, pp. 383-397, 2018.

[7] A.A. Bashlykova, A.A. Zatsarinny, A.A. Kamenschikov, S.V. Kozlov, A.Y. Oleinikov and I.I. Chusov, "Interoperability as a scientific and legal basis of a seamless integration of information-telecommunication systems" in Systems and means of informatics, Moscow:Torus-Press, vol. 8, no. 4, pp. 61-72, 2018.

[8] S.V. Kozlov, S.I. Makarenko, A. Ya. Oleynikov, D.V. Rastiagaev and T.E. Chernitskaya, "The Problem of interoperability in network-centric control systems", *Journal of radio electronics [electronic journal]*, no. 12, 2019.

[9] E.N. Evdokimova, "Evolution of the process approach in management and prospects for its development", *Management of economic systems: Electronic scientific journal*, vol. 28, no. 4, pp. 117-124, 2011.
[10] S.V. Kozlov and A.N. Kubankov, "The evolution of the methods of the process approach to the development of automated management information systems", *Quality. Innovations. Education*, vol. 156, no. 5, pp. 103-110, 2018.