Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Операционные системы

Студент: Лопатнюк П.В.

ФИТ 3 курс 1 группа

Преподаватель: Бернацкий П.В.

Минск 2024

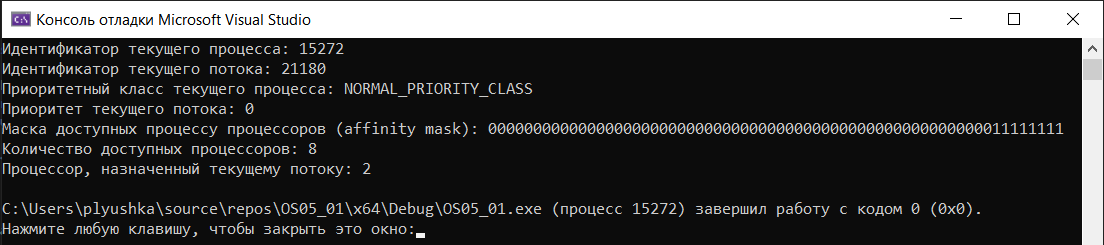
**Лабораторная работа 05**

**Задание 01**

1. Разработайте консольное Windows-приложение **OS05\_01** на языке С++, выводящее на консоль следующую информации:

* идентификатор текущего процесса;
* идентификатор текущего (main) потока;
* приоритет (приоритетный класс) текущего процесса;
* приоритет текущего потока;
* маску (affinity mask) доступных процессу процессоров в двоичном виде;
* количество процессоров, доступных процессу;
* процессор, назначенный текущему потоку.

|  |
| --- |
| #include <iostream>  #include <Windows.h>  #include <bitset>  using namespace std;  void printBinaryMask(DWORD\_PTR mask) {  for (int i = sizeof(mask) \* 8 - 1; i >= 0; i--) {  cout << ((mask >> i) & 1);  }  cout << endl;  }  int main() {  setlocale(LC\_ALL, "rus");  // Дескриптор текущего процесса и потока  HANDLE hProcess = GetCurrentProcess();  HANDLE hThread = GetCurrentThread();  // Идентификаторы процесса и потока  DWORD processId = GetCurrentProcessId();  DWORD threadId = GetCurrentThreadId();  // Приоритетный класс процесса  DWORD processPriorityClass = GetPriorityClass(hProcess);  int threadPriority = GetThreadPriority(hThread);  // Маска доступных процессоров и количество процессоров  DWORD\_PTR processAffinityMask, systemAffinityMask;  GetProcessAffinityMask(hProcess, &processAffinityMask, &systemAffinityMask);  DWORD numberOfProcessors = bitset<sizeof(DWORD\_PTR) \* 8>(processAffinityMask).count();  // Процессор, назначенный текущему потоку  DWORD\_PTR threadIdealProcessor = SetThreadIdealProcessor(hThread, MAXIMUM\_PROCESSORS);  // Вывод информации  cout << "Идентификатор текущего процесса: " << processId << endl;  cout << "Идентификатор текущего потока: " << threadId << endl;  cout << "Приоритетный класс текущего процесса: ";  switch (processPriorityClass) {  case IDLE\_PRIORITY\_CLASS: cout << "IDLE\_PRIORITY\_CLASS"; break; //64  case BELOW\_NORMAL\_PRIORITY\_CLASS: cout << "BELOW\_NORMAL\_PRIORITY\_CLASS"; break;//16384  case NORMAL\_PRIORITY\_CLASS: cout << "NORMAL\_PRIORITY\_CLASS"; break; //32  case ABOVE\_NORMAL\_PRIORITY\_CLASS: cout << "ABOVE\_NORMAL\_PRIORITY\_CLASS"; break;//32768  case HIGH\_PRIORITY\_CLASS: cout << "HIGH\_PRIORITY\_CLASS"; break; //128  case REALTIME\_PRIORITY\_CLASS: cout << "REALTIME\_PRIORITY\_CLASS"; break; //256  default: cout << "UNKNOWN"; break;  }  cout << endl;  cout << "Приоритет текущего потока: " << threadPriority << endl;  cout << "Маска доступных процессу процессоров (affinity mask): ";  printBinaryMask(processAffinityMask);  cout << "Количество доступных процессоров: " << numberOfProcessors << endl;  cout << "Процессор, назначенный текущему потоку: " << threadIdealProcessor << endl;  return 0;  } |

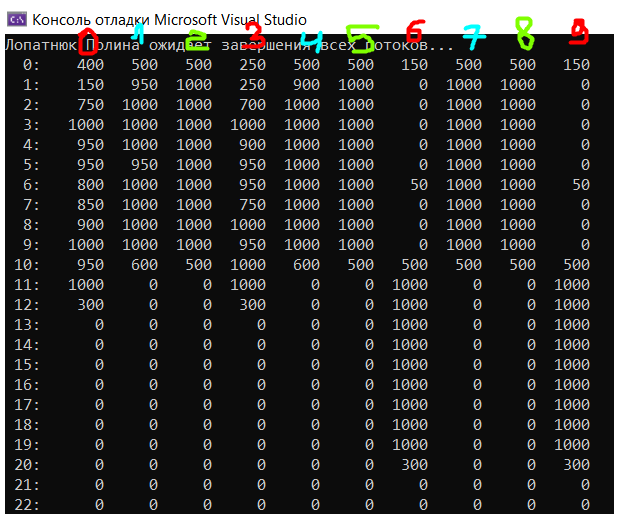


**Задание 02**

2. Создайте консольное Windows **OS05\_02** на языке С#**,** взяв за основу приложение **OS04\_07** из Лабораторной работы №4. Измените метод Main таким образом, чтобы потоки 0, 3, 6 и т.д. запускались с минимальным приоритетом потока, а потоки 2, 5, 8... – с максимальным. Класс приоритета процесса оставьте по умолчанию (Normal). Как пример, можно использовать следующий фрагмент кода.

|  |
| --- |
| public class Program  {  const int ThreadCount = 10; // Количество потоков  const int ThreadLifeTime = 10; // Время работы каждого потока в секундах  const int ObservationTime = 30; // Время наблюдения в секундах  static int[,] Matrix = new int[ThreadCount, ObservationTime];  static DateTime StartTime = DateTime.Now;  static void WorkThread(object o)  {  int id = (int)o;  for (int i = 0; i < ThreadLifeTime \* 20; i++)  {  DateTime CurrentTime = DateTime.Now;  int ElapsedSeconds = (int)Math.Round(CurrentTime.Subtract(StartTime).TotalSeconds);  if (ElapsedSeconds < ObservationTime)  {  Matrix[id, ElapsedSeconds] += 50;  }  MySleep(50);  }  }  public static void MySleep(int milliseconds)  {  var startTime = DateTime.Now;  while (DateTime.Now - startTime < TimeSpan.FromMilliseconds(milliseconds))  {  int result = 1;  for (int i = 0; i < 10000; i++)  {  result \*= 2;  }  }  }  static void Main(string[] args)  {  Thread[] threads = new Thread[ThreadCount];  for (int i = 0; i < ThreadCount; ++i)  {  object o = i;  threads[i] = new Thread(WorkThread);  // Устанавливаем приоритет в зависимости от индекса потока  if (i % 3 == 0)  {  threads[i].Priority = ThreadPriority.Lowest; // Индексы 0, 3, 6, 9  }  else if (i % 3 == 2)  {  threads[i].Priority = ThreadPriority.Highest; // Индексы 2, 5, 8  }  // Иначе — приоритет по умолчанию (Normal)  threads[i].Start(o);  }  Console.WriteLine("Лопатнюк Полина ожидает завершения всех потоков...");  for (int i = 0; i < ThreadCount; ++i)  {  threads[i].Join();  }  for (int s = 0; s < ObservationTime; s++)  {  Console.Write("{0,3}: ", s);  for (int th = 0; th < ThreadCount; th++)  {  Console.Write(" {0,5}", Matrix[th, s]);  }  Console.WriteLine();  }  }  } |

3. Выполните приложение, не забудьте про персонификацию вывода результатов.



4. По зафиксированным скриншотам объясните полученные результаты.

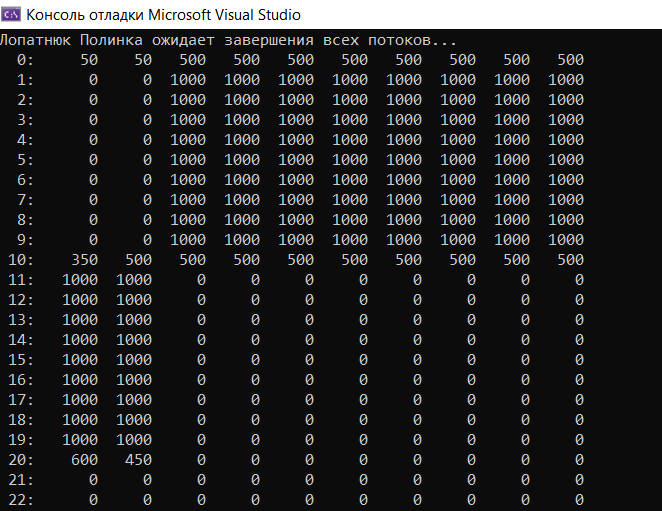
Потоки с низким приоритетом (например, 0, 3, 6, установленные на ThreadPriority.Lowest) обновляют матрицу медленнее и имеют более низкие значения (например, 100–500), так как они получают меньше процессорного времени по сравнению с высокоприоритетными потоками.

**Задание 03**

5. Создайте консольное Windows **OS05\_03** на языке С#, взяв за основу приложение **OS05\_02** из настоящей работы. На этот раз только несколько потоков запустите на наименьшем приоритете потока, а остальные – на наибольшем.

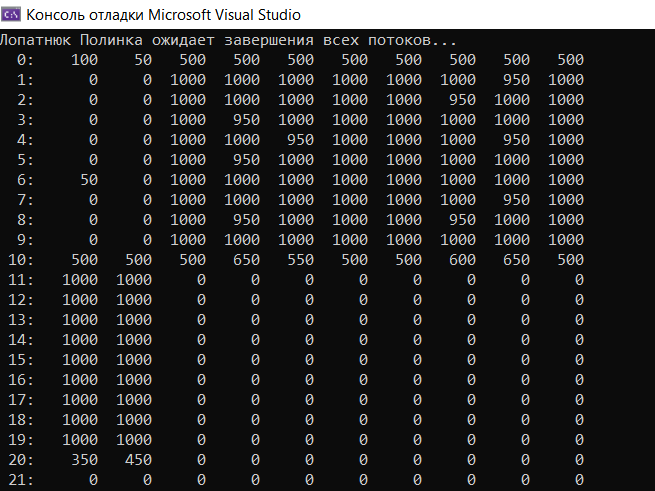
|  |
| --- |
| for (int i = 0; i < ThreadCount; ++i)  {  object o = i;  threads[i] = new Thread(WorkThread);  // Устанавливаем приоритет в зависимости от индекса потока  if (i < 2) // Устанавливаем для первых двух потоков наименьший приоритет  {  threads[i].Priority = ThreadPriority.Lowest;  }  else // Для остальных потоков — наибольший приоритет  {  threads[i].Priority = ThreadPriority.Highest;  }  threads[i].Start(o);  } |

6. Выполните приложение **OS05\_03**. Удалось ли поработать низкоприоритетным потокам? (Чтобы уменьшить влияние случайности, можно повторить эксперимент несколько раз).



7. Выполните приложение **OS05\_03** с другими парами приоритетов, например, BelowNormal и Normal. Изменился ли характер работы потоков?

|  |
| --- |
| for (int i = 0; i < ThreadCount; ++i)  {  object o = i;  threads[i] = new Thread(WorkThread);  // Устанавливаем приоритет в зависимости от индекса потока  if (i < 2) // Первые два потока с приоритетом BelowNormal  {  threads[i].Priority = ThreadPriority.BelowNormal;  }  else // Остальные потоки с приоритетом Normal  {  threads[i].Priority = ThreadPriority.Normal;  }  threads[i].Start(o);  } |



8. По зафиксированным скриншотам объясните полученные результаты. При этом укажите числовые значения приоритетов потоков.

ThreadPriority.Lowest = 1(наименьший приоритет)

ThreadPriority.BelowNormal = 2 (ниже нормального)

ThreadPriority.Normal = 3 (по умолчанию)

ThreadPriority.AboveNormal = 4 (выше нормального)

ThreadPriority.Highest = 5 (наибольший приоритет)

При запуске программы с потоками, установленными на разные приоритеты, операционная система будет решать, какие потоки должны выполняться в первую очередь, в зависимости от их приоритета и доступных ресурсов. Реальная разница в исполнении будет зависеть от множества факторов, таких как:

* Нагрузка на процессор.
* Частота переключения контекста.
* Реализация планирования операционной системы

Потоки с приоритетом Lowest выполняют свою работу медленнее и реже, потому что в системе есть потоки с более высоким приоритетом. Это может приводит к тому, что их выполнение отстаёт.

Потоки с приоритетом Highest получают процессорное время чаще и завершают свою работу быстрее.

Для первых двух потоков установлены приоритеты BelowNormal. Эти потоки получают процессорное время реже, чем потоки с приоритетом Normal, и в условиях сильной загрузки процессора они отстают.

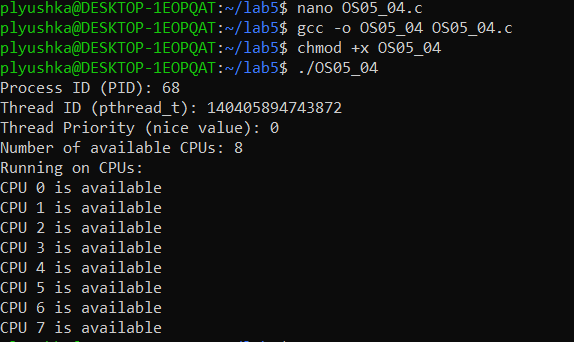
Для остальных потоков установлены приоритеты Normal. Эти потоки получат процессорное время чаще, чем потоки с приоритетом BelowNormal, но не так часто, как потоки с более высоким приоритетом.

**Задание 04**

9. Разработайте консольное Linux-приложение **OS05\_04** на языке С++, выводящее на консоль следующую информации:

* идентификатор текущего процесса;
* идентификатор текущего (main) потока;
* приоритет (nice) текущего потока;
* номера доступных процессоров.

|  |
| --- |
| #define \_GNU\_SOURCE  #include <stdio.h>  #include <unistd.h>  #include <pthread.h>  #include <sys/resource.h>  #include <sched.h>  #include <errno.h>  int main() {  // Получаем идентификатор текущего процесса  pid\_t pid = getpid();  printf("Process ID (PID): %d\n", pid);  // Получаем идентификатор текущего потока  pthread\_t tid = pthread\_self();  printf("Thread ID (pthread\_t): %lu\n", (unsigned long)tid);  // Получаем приоритет (nice) текущего потока  int priority = getpriority(PRIO\_PROCESS, 0); // 0 — это ID текущего процесса  printf("Thread Priority (nice value): %d\n", priority);  // Получаем количество доступных процессоров  int num\_cpus = sysconf(\_SC\_NPROCESSORS\_ONLN); // Получаем количество доступных процессоров  printf("Number of available CPUs: %d\n", num\_cpus);  // Инициализируем набор процессоров  cpu\_set\_t cpu\_set;  CPU\_ZERO(&cpu\_set);  // Получаем информацию о процессах, к которым привязан текущий процесс  if (sched\_getaffinity(0, sizeof(cpu\_set), &cpu\_set) == -1) {  perror("sched\_getaffinity");  return 1;  }  printf("Running on CPUs:\n");  for (int i = 0; i < num\_cpus; i++) {  // Проверяем, на каком процессоре может работать текущий процесс  if (CPU\_ISSET(i, &cpu\_set)) {  printf("CPU %d is available\n", i);  }  }  return 0;  } |

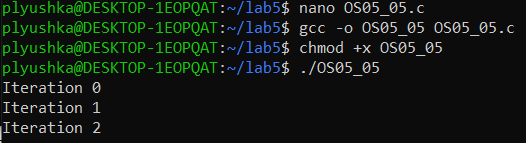


**Задание 05**

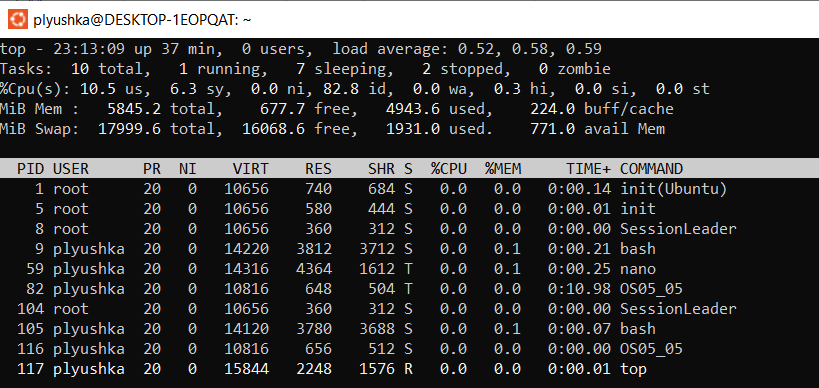
10. Разработайте консольное Linux-приложение **OS05\_05** на языке С, выполняющее длинный цикл.

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  int main() {  for (int i = 0; i < 10; i++) {  printf("Iteration %d\n", i);  sleep(1);  }  return 0;  } |

11. Запустите приложение **OS05\_05.**

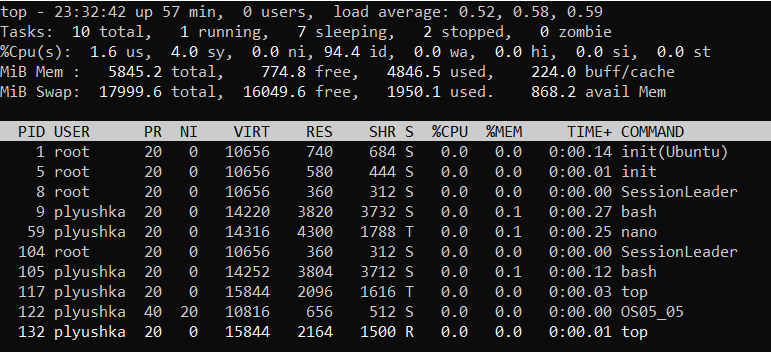


12. Зафиксируйте текущее значение **nicе**, полученное с помощью команды **top**.



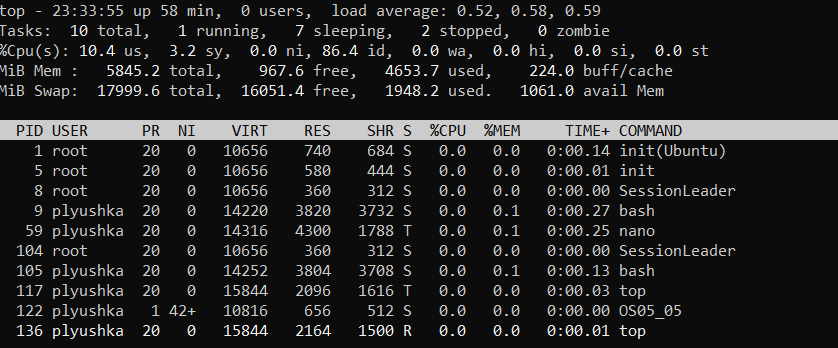
13. Увеличьте приоритет для **OS05\_05** до максимального значения (самого привилегированного). Зафиксируйте текущее значение **nicе**, полученное с помощью команды **top**.





14. Уменьшите приоритет для **OS05\_05** до минимального значения (самого ничтожного). Зафиксируйте текущее значение **nicе**, полученное с помощью команды **top**





**Задание 06.** Ответьте на следующие вопросы

**15. Поясните понятие «мультизадачная OS с вытеснением».**

**Мультизадачная ОС** позволяет нескольким программам (или процессам) выполняться одновременно на одном компьютере.

**Вытесняющая многозадачность** — это тип многозадачности, где операционная система (ОС) активно управляет выполнением процессов, «вытесняя» один процесс, чтобы дать возможность работать другому.

**Основные принципы:**

* **Квантование времени ЦП:** ОС выделяет каждому процессу ограниченный отрезок времени (квант времени) для работы на процессоре. По истечении кванта времени ОС прерывает работу текущего процесса и переключает процессор на другой процесс.
* **Приоритеты потоков:** Каждому процессу (или потоку) присваивается приоритет. Процессы с более высоким приоритетом получают большее количество времени на процессоре, а процессы с низким приоритетом могут ждать своей очереди.

**16. Поясните понятие «циклическое планирование».**

**Циклическое планирование** — это алгоритм планирования задач, который предоставляет каждому процессу определенное количество времени для выполнения на процессоре. По истечении кванта времени, процесс переходит в конец очереди и ждет своей очереди снова.

**Преимущества:**

* Простота реализации: Алгоритм легко реализовать и поддерживать.
* Справедливость: Каждый процесс получает равное время на процессоре.
* Хорошо подходит для интерактивных систем: Предоставляет возможность быстро реагировать на запросы пользователя, даже если другие процессы выполняются.

**Недостатки:**

* Низкая эффективность при большом количестве процессов: Частые переключения контекста могут занимать значительное время, что снижает производительность.
* Неоптимальное для процессов с разными требованиями: Не учитывает приоритеты процессов или их потребности в ресурсах.

**17. Поясните понятие «приоритетное планирование».**

**Приоритетное планирование** — это метод управления процессами, при котором каждому процессу назначается приоритет, и первыми запускаются процессы с более высоким приоритетом. Это учитывает срочность задач и оптимизирует использование ресурсов, особенно для процессов, зависящих от операций ввода-вывода.

Приоритеты могут быть статическими или динамическими: статические не меняются, а динамические корректируются в зависимости от активности процесса. Например, процессы, которые меньше занимают процессор, получают более высокий приоритет, что ускоряет их обработку.

Процессы также могут группироваться по классам приоритетности, где каждый класс получает ресурсы в порядке своего приоритета, а внутри класса действует циклическое планирование.

**18. Поясните понятие «кооперативное планирование».**

**Кооперативное планирование** — это метод управления задачами, при котором операционная система переключает управление между задачами только тогда, когда задача добровольно завершает свою работу или уступает управление. В этом случае ОС не может принудительно прервать процесс, что делает её зависимой от корректного поведения приложений. Если задача не уступит управление, остальные задачи будут вынуждены ждать, что может снизить общую производительность и отзывчивость системы.

**19. Поясните понятие «OS реального времени».**

**Операционная система реального времени (RTOS)** — это ОС, обеспечивающая выполнение задач в заданные временные интервалы с предсказуемым временем отклика на внешние события. Такие системы применяются там, где критичны задержки в обработке данных, например, в медицинских приборах, промышленной автоматике и авионике. RTOS минимизирует время реакции, гарантируя выполнение задач точно по расписанию, чтобы избежать сбоев и обеспечить стабильность работы критически важных систем.

**20. Поясните понятие «приоритет процесса».**

**Приоритет процесса** — это значение, которое определяет, в каком порядке операционная система будет распределять ресурсы (в частности, процессорное время) между процессами. Процессы с более высоким приоритетом получают доступ к процессору раньше, что позволяет управлять их очередностью выполнения в зависимости от важности.

**21. Поясните выражение «поток уступает процессор другому потоку».**

Выражение «поток уступает процессор другому потоку» означает, что текущий поток завершает своё выполнение или приостанавливается, чтобы освободить процессор для выполнения другого потока. Это может происходить как добровольно, когда поток сам решает уступить управление (например, при завершении своей задачи или в кооперативном планировании), так и принудительно, когда операционная система решает приостановить поток, чтобы предоставить ресурсы другому процессу (например, в случае приоритетного планирования или тайм-слота).

**22. Windows: как поток может уступить процессор?**

1. **Явное освобождение процессора:** Поток может вызвать системную функцию SwitchToThread(), чтобы добровольно уступить процессор другому потоку с одинаковым приоритетом. Это позволяет текущему потоку передать управление другим потокам, не завершая выполнение.

2. **Ожидание:** Поток может вызвать операции, которые заставляют его приостановиться до выполнения какого-либо события или условия, например:

* Ожидание на синхронизацию с другими потоками через механизмы, такие как WaitForSingleObject(), WaitForMultipleObjects() или Sleep().
* Ожидание ввода-вывода или других ресурсов, что временно освобождает процессор для других потоков.

1. **Завершение работы:** Когда поток завершает выполнение (например, через ExitThread() или завершение функции, которая была основной для потока), процессор освобождается и может быть передан другому потоку.
2. **Прерывание:** В многозадачных системах Windows, где используется приоритетное планирование, операционная система может принудительно приостановить выполнение потока с меньшим приоритетом и переключиться на поток с более высоким приоритетом, даже если текущий поток ещё не завершил свою работу.

**23. Windows: что такое базовый приоритет потока, как он вычисляется и диапазон его изменения?**

**Базовый приоритет потока** — это начальный приоритет, с которым поток запускается в операционной системе Windows. Этот приоритет определяется в зависимости от приоритета самого процесса и класса приоритетов, к которому принадлежит поток.

Каждый процесс в Windows имеет свой базовый приоритет, который влияет на приоритет всех потоков этого процесса. Когда поток создается, его базовый приоритет начинается с базового приоритета процесса.

Windows использует различные классы приоритетов для потоков:

* Real-time (реального времени) — самый высокий приоритет.
* High (высокий) — выше, чем у нормальных потоков.
* Above Normal (выше нормального) — немного выше нормального.
* Normal (нормальный) — стандартный приоритет для большинства потоков.
* Below Normal (ниже нормального) — немного ниже нормального.
* Idle — самый низкий приоритет, используется для потоков, выполняющихся, когда нет других задач.

**24. Windows: поясните назначение и принцип применения системного вызова ResumeThread.**

Функция ResumeThread в Windows возобновляет выполнение приостановленного потока. Она уменьшает счетчик приостановки на единицу каждый вызов. Поток возобновляется, когда счетчик приостановки достигает нуля.

hThread — дескриптор потока, который нужно возобновить.

При успешном выполнении возвращается предыдущее значение счетчика приостановки.

При ошибке возвращается (DWORD)-1.

Эта функция работает в паре с SuspendThread, которая увеличивает счетчик приостановки.

**25. Windows: поясните назначение и принцип применения системного вызова WaitForSingleObject.**

Функция WaitForSingleObject в Windows блокирует выполнение текущего потока и ожидает, пока указанный объект не перейдет в сигнальное состояние, либо пока не истечет указанный срок ожидания.

Параметры:

* hHandle — идентификатор объекта, который должен быть в сигнальном состоянии (например, мьютекс, событие, семафор и т.д.).
* dwMilliseconds — время ожидания в миллисекундах:
* Если INFINITE — функция будет ожидать неограниченно.
* Если 0 — функция немедленно возвращает управление, не блокируя поток, проверяя состояние объекта.

Возвращаемое значение:

* WAIT\_ABANDONED — поток, владевший объектом, завершился без перевода объекта в сигнальное состояние.
* WAIT\_OBJECT\_0 — объект перешел в сигнальное состояние.
* WAIT\_TIMEOUT — истек срок ожидания.
* WAIT\_FAILED — ошибка (неверный идентификатор объекта, вызов GetLastError() дает подробности).

Эта функция используется для синхронизации потоков, когда один поток должен ожидать, пока другой завершит операцию или установит объект в сигнальное состояние.

**26. Windows: поясните назначение и принцип применения системных вызовов GetProcessPriorityBoost, GetThreadPriorityBoost, SetProcessPriorityBoost, SetThreadPriorityBoost.**

Функции GetProcessPriorityBoost, GetThreadPriorityBoost, SetProcessPriorityBoost и SetThreadPriorityBoost управляют динамическим повышением приоритетов процессов и потоков в Windows. Они позволяют включать или отключать автоматическое повышение приоритетов, которое обеспечивает систему, чтобы гарантировать отзывчивость и предотвращение голодания потоков.

Динамическое повышение приоритета: Система может повышать приоритет потоков с базовым приоритетом от 1 до 15 (для потоков с приоритетом от 16 до 31 повышение не происходит), чтобы гарантировать, что они получают достаточный процессорный ресурс.

С помощью этих функций можно включать или отключать эту функцию для отдельных потоков или процессов.

**1. SetThreadPriorityBoost:** Отключает или включает динамическое повышение приоритета для конкретного потока.

Параметры:

* hThread: дескриптор потока.
* bDisablePriorityBoost: если TRUE, динамическое повышение отключается; если FALSE, оно включается.

Возвращаемое значение: TRUE при успешном выполнении, FALSE в случае ошибки.

**2. SetProcessPriorityBoost:** Отключает или включает динамическое повышение приоритета для всех потоков в процессе.

Параметры:

* hProcess: дескриптор процесса.
* bDisablePriorityBoost: если TRUE, динамическое повышение отключается; если FALSE, оно включается.

Возвращаемое значение: TRUE при успешном выполнении, FALSE в случае ошибки.

**3. GetThreadPriorityBoost:** Проверяет, отключено ли динамическое повышение приоритета для конкретного потока.

Параметры:

* hThread: дескриптор потока.
* pDisablePriorityBoost: указатель на переменную, которая получает состояние функции.

Возвращаемое значение: TRUE при успешном выполнении, FALSE в случае ошибки.

**4. GetProcessPriorityBoost:** Проверяет, отключено ли динамическое повышение приоритета для всех потоков в процессе.

Параметры:

* hProcess: дескриптор процесса.
* pDisablePriorityBoost: указатель на переменную, которая получает состояние функции.

Возвращаемое значение: TRUE при успешном выполнении, FALSE в случае ошибки.

**27. Linux: поясните принцип идентификации процессов и потоков и поясните, почему он такой.**

В Linux процессы и потоки идентифицируются с помощью уникальных числовых идентификаторов. Каждый процесс имеет уникальный идентификатор, который называется PID (Process ID). Этот идентификатор является целым числом в диапазоне от 0 до 32767. PID 0 зарезервирован для процесса ядра (swapper), а PID 1 — для init-процесса (или его аналога, например, systemd), который является родительским для всех остальных процессов. Каждый процесс имеет только один PID, и этот идентификатор сохраняется неизменным на протяжении всего жизненного цикла процесса.

Для потоков используется идентификатор TID (Thread ID), который уникально определяет поток внутри процесса. В однопоточном процессе TID совпадает с PID, потому что процесс состоит только из одного потока. Однако в многопоточном процессе каждый поток получает свой уникальный TID, но все потоки одного процесса имеют одинаковый PID.

Такой принцип идентификации обусловлен тем, что процесс является независимой единицей исполнения, которая имеет свои ресурсы, такие как память и файловые дескрипторы. Потоки же являются более легкими единицами исполнения, которые разделяют ресурсы процесса. Использование разных идентификаторов для процессов и потоков позволяет операционной системе эффективно управлять ими, а также упрощает задачу отслеживания и контроля их состояния. В случае с однопоточными программами совпадение TID с PID упрощает управление, так как для единственного потока не требуется отдельный идентификатор.

**28. Linux: Поясните понятие «планировщик потоков».**

Планировщик потоков в Linux — это компонент ядра, который управляет распределением времени процессора между различными процессами и потоками. Он решает, какой процесс или поток должен быть выполнен в данный момент, основываясь на различных факторах, таких как приоритет, значение nice (параметр для регулирования приоритета процесса), политика планирования и другие параметры, влияющие на выполнение. В Linux потоки рассматриваются как процессы, но с общими ресурсами, такими как память и файловые дескрипторы. Поэтому планировщик работает с потоками так же, как и с процессами, обеспечивая эффективное использование процессорного времени и справедливое выполнение всех задач в системе.

**29. Linux: поясните принцип использования значения nice – процесса, диапазон его изменения, для какого режима работы планировщика это значение применяется?**

Значение **nice** в Linux — это параметр, который позволяет пользователю задавать предпочтительный приоритет процесса. Это значение влияет на приоритет, с которым процесс будет получать процессорное время, но не является прямым назначением приоритета в планировщике. Планировщик процессов преобразует значение nice в статический приоритет, который используется для управления временем процессора.

Диапазон значений **nice** составляет от **-20** до **19**. Чем ниже значение nice, тем выше приоритет процесса. Например, значение **-20** означает самый высокий приоритет, а значение **19** — самый низкий. При этом планировщик процессов использует более широкий диапазон приоритетов от **0** до **139**. Для процессов с заданным значением nice от **-20** до **+19**, приоритеты в планировщике будут варьироваться от **100** (для наивысшего nice) до **139** (для самого низкого nice).

Значение **nice** применяется в **обычном режиме работы планировщика** (когда процесс не требует реального времени), где оно влияет на то, как часто и сколько процесс будет получать доступ к процессору. В реальном времени (с приоритетом 0) значение nice не влияет, так как процессы реального времени получают приоритет над всеми остальными.

**30. Linux: перечислите политики планирования, какая действует по умолчанию?**

SCHED\_OTHER (или стандартная, разделения времени) — это политика по умолчанию для обычных (не привилегированных) процессов. Приоритет таких процессов определяется параметром nice, и их выполнение происходит, если нет процессов с более высокоприоритетными политиками, как FIFO или RR.

SCHED\_BATCH — применяется для длительных фоновых вычислений. В отличие от SCHED\_OTHER, эта политика снижает частоту переключений задач, что оптимизирует выполнение задач, не требующих быстрого отклика.

SCHED\_FIFO (FIFO) — используется для процессов реального времени. В этой политике каждый процесс имеет фиксированный статический приоритет, и процесс с более высоким приоритетом выполняется первым. Процесс работает до тех пор, пока не заблокируется или не уступит процессор по вызову sched\_yield().

SCHED\_RR (Round Robin) — также используется для процессов реального времени, но с квантованием времени. Каждый процесс получает фиксированный квант времени и, по его истечении, переходит в конец своей очереди, позволяя другим процессам с таким же приоритетом выполнять свою работу по очереди.

По умолчанию в Linux используется политика SCHED\_OTHER, которая подходит для большинства обычных процессов.

**31. Linux: с помощью какого системного вызова поток может уступить процессор.**

В Linux поток может уступить процессор, вызвав системный вызов sched\_yield. Этот вызов сообщает ядру, что текущий процесс (или поток) добровольно уступает процессорное время, перемещая себя в конец очереди процессов с таким же приоритетом. После этого управление передается следующему доступному процессу, который имеет такой же приоритет. Это позволяет системе эффективно распределять процессорное время между процессами с одинаковыми приоритетами.

