Теория и практика многопоточного программирования

Семинар 2

Неганов Алексей

Московский физико-технический институт (национальный исследовательский университет)
Кафедра теоретической и прикладной информатики

Москва 2020

Рисунок 1 – Упрощённая диаграмма состояний процесса

Процессы

Что содержит процесс (пример UNIX):

- Сегмент кода (возможно, разделяемый)
- Сегмент данных (включая стек)
- Состояние регистров, включая программный счётчик
- Таблица дескрипторов ввода-вывода
- Диспозиция обработки сигналов
- Счётчики потреблённых ресурсов
- Командная строка и окружение
- Текущий и корневой каталог
- Идентификаторы процесса, его родителя, группы и сеанса
- Информация о владельце
- Настройки прав
- ...

Что содержит поток:

- Стек
- Состояние регистров, включая программный счётчик

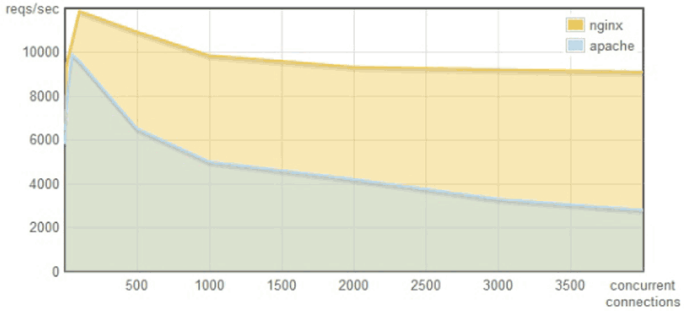

Рисунок 2 – Сравнение поведения веб-серверов Nginx и Apache при увеличении количества обновременных подключений

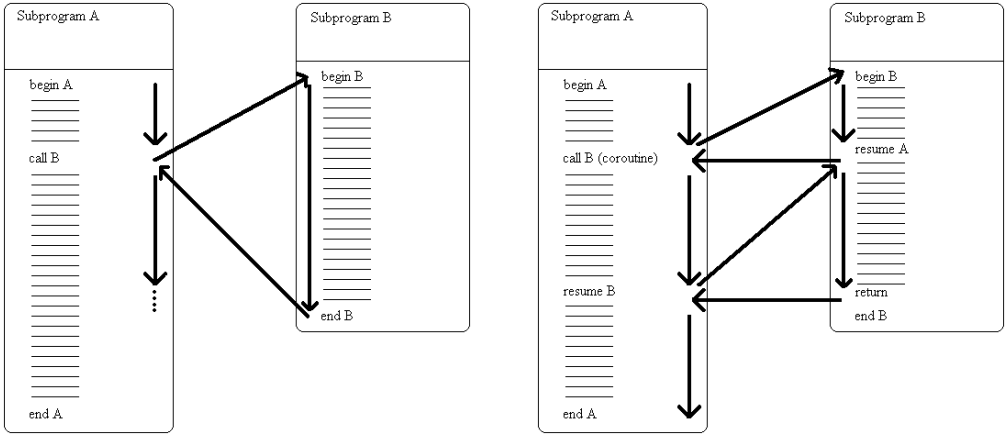

Рисунок 3 – Поток команд для программ и сопрограмм

Green threads / goroutines

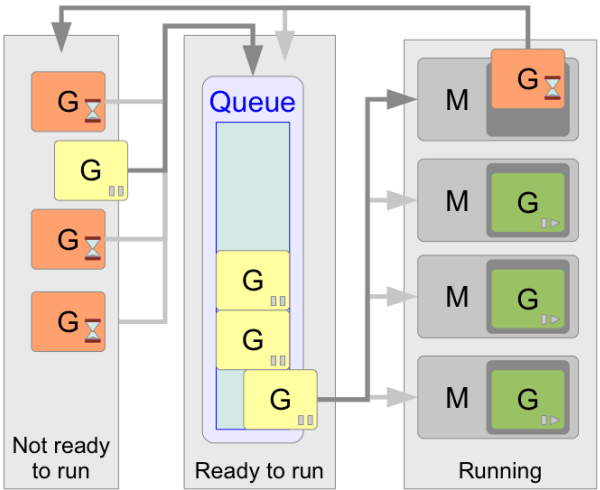

Рисунок 4 – Схема работы планировщика Go

Ситуация гонок

Ситуацией гонок *(race condition)* называется недетерминированность совокупного исполнения потоков.

Критерий Бернстайна

Пусть $R(P_i)$ множество переменных, значение которых поток P_i использует в операциях чтения, $W(P_i)$ — множество переменных, использующихся в операциях записи. Тогда совокупное исполнение потоков P_1 и P_2 детерминировано, если

$$\begin{cases} W(P_1) \cap W(P_2) = \varnothing \\ R(P1) \cap W(P_2) = \varnothing \\ W(P_1) \cap R(P_2) = \varnothing \end{cases} \quad (1)$$

Условия на код

1. Два и более потока не должны ни при каких условиях находиться одновременно в критических секциях, связанных с одними и теми же данными *(mutual exclusion)*
2. Хотя бы один из конкурирующих потоков должен рано или поздно попасть в нужную ему критическую секцию *(freedom from deadlock)*
3. Каждый поток должен рано или поздно попасть в критическую секцию *(freedom from starvation)*
4. Полезная работа над данными критической секции должна доводиться до конца *(freedom from livelock)*
5. Основное время поток должен провести в исполнении полезной работы, а не в ожидании блокированного ресурса *(freedom from lock contention)*
6. Заблокированный поток не должен расходовать процессорное время (отсутствие активного ожидания)

Условия на код

Верно ли, что методы `lock12()` и `lock21()`, будучи вызванными из разных потоков, рано или поздно завершат работу? Какое название имеет такое состояние программы?

```
mutex_t m1, m2;

void lock12(void) {
      m1.lock();
      while (m2.locked()) {
            m1.unlock();
            wait();
            m1.lock();
      }
      m2.lock();
}
```

```
void lock21(void) {
      m2.lock();
      while (m1.locked()) {
            m2.unlock();
            wait();
            m2.lock();
      }
      m1.lock();
}
```

Необходимые и достаточные условия возникновения тупиков (deadlock)

1. Условие взаимоисключения *(mutual exclusion):* существует ресурс такой, что единовременно использовать его только один поток
2. Условие ожидания ресурсов *(hold and wait):* потоки, уже захватившие ресурсы, могут захватывать другие ресурсы
3. Условие неперераспределяемости *(no preemption):* ресурс не может быть принудительно забран у потока, только сам поток может освободить его
4. Условие кругового ожидания *(circular wait):* существует кольцевая цепь потоков, в которой каждый поток ждет доступа к ресурсу, удерживаемому другим потоком цепи

Необходимые и достаточные условия возникновения тупиков (deadlock)

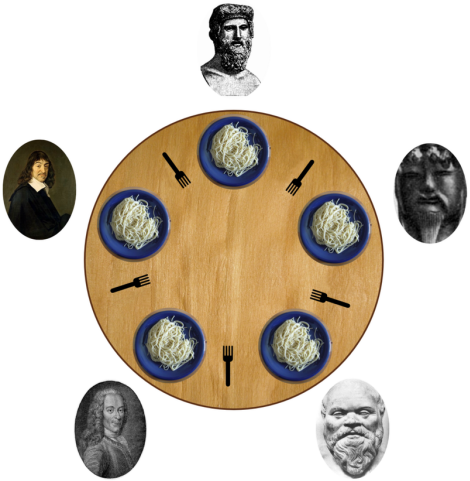

Рисунок 5 – Задача об обедающих философах

Необходимые и достаточные условия возникновения тупиков (deadlock)

Рассмотрим псевдокод следующей программы, использующей семафоры:

```
int x = 0, y = 0, z = 0;
sem lock1 = 1, lock2 = 1;

process foo {                    process bar {
      z = z + 2;                       P(lock2);
      P(lock1);                        y = y + 1;
      x = x + 2;                       P(lock1);
      P(lock2);                        x = x + 1;
      V(lock1);                        V(lock1);
      y = y + 2;                       V(lock2);
      V(lock2);                        z = z + 1;
}                                }
```

Может ли эта программа войти в состояние взаимной блокировки *(deadlock)?* Если да, то каким образом и каковы возможные значения переменных x, y и z в этом состоянии?

Необходимые и достаточные условия возникновения тупиков (deadlock)

Рассмотрим следующий код на языке C:

```
char buf[100];
int rc;
int fd[2];
pipe(fd);
if (fork() == 0) {
   dup2(fd[1], 1);
   close(fd[1]);
   close(fd[0]);
   execlp("ls", "ls", NULL);
   perror("ls");
   exit(1);
}
close(fd[1]);
wait(NULL);
while((rc = read(fd[0], buf, sizeof(buf)))>0) {
   /* ... */
}
```

Может ли эта программа войти в состояние взаимной блокировки *(deadlock)?* Если да, то каким образом?

Задачи

1. Напишите программу, принимающую число *N* как аргумент командной строки и запускающую *N* потоков с помощью библиотеки POSIX threads (`pthreads`) или C++11 threads. Каждый поток должен напечатать число — номер в порядке запуска и свой ID.
2. Напишите программу с использованием `pthreads` или C++11 threads, где 4 потока заполняют промежуточный буфер (единый для всех потоков) символами `'1'`, `'2'`, `'3'` или `'4'` соответственно, причём после 100 символов поток заканчивает свою работу, и ещё 4 потока читают данные из буфера и пишут в 4 файла, причём каждый поток должен считать из буфера 100 символов.
3. Напишите исправленную версию программы, вызывающей `ls` в дочернем процессе.