Вспоминаем дискретную математику

- Что такое множество / подмножество?
- Что такое бинарное отношение на множестве?
- Что такое (ир)рефлексивность?
- Что такое (анти)симметричность?
- Что такое транзитивность?
- Что такое отношение эквивалентности?
- Что такое отношение (строгого) порядка?
- Что такое линейный / частичный порядок?
- Что такое сравнимые / несравнимые элементы?

Повторяем материал

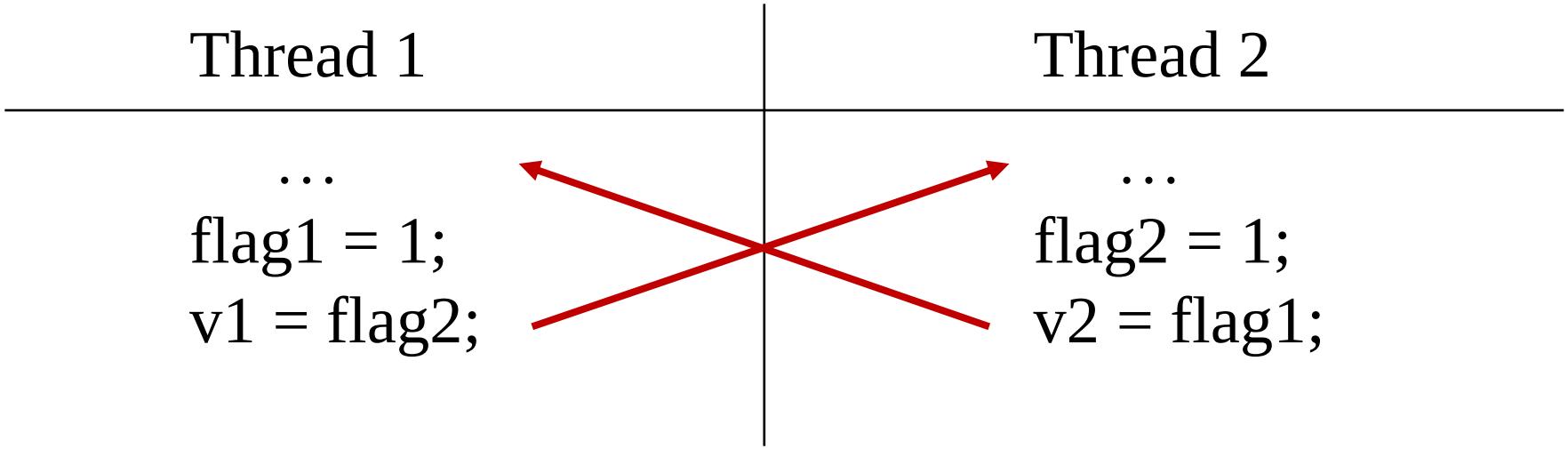
Изначально flag1 == flag2 ==0

Thread 1	Thread 2
flag1 = 1; v1 = flag2;	flag2 = 1; v2 = flag1;

v1 == ? v2 ==?

Повторяем материал

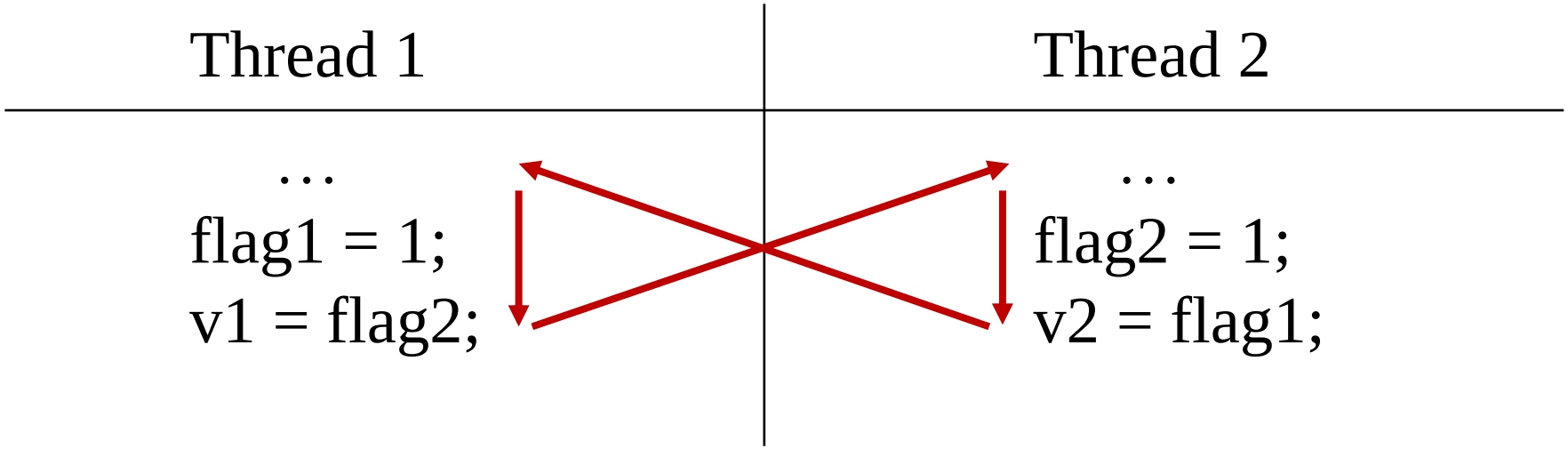
Изначально flag1 == flag2 ==0



v1 == v2 ==0 ?

Повторяем материал

Изначально flag1 == flag2 ==0



v1 == v2 ==0 ?

Повторяем материал

Изначально flag1 == flag2 ==0

Thread 1	Thread 2
flag1 = 1; v1 = flag2; ←	flag2 = 1; v2 = flag1; ←

v1 == v2 ==0 ?

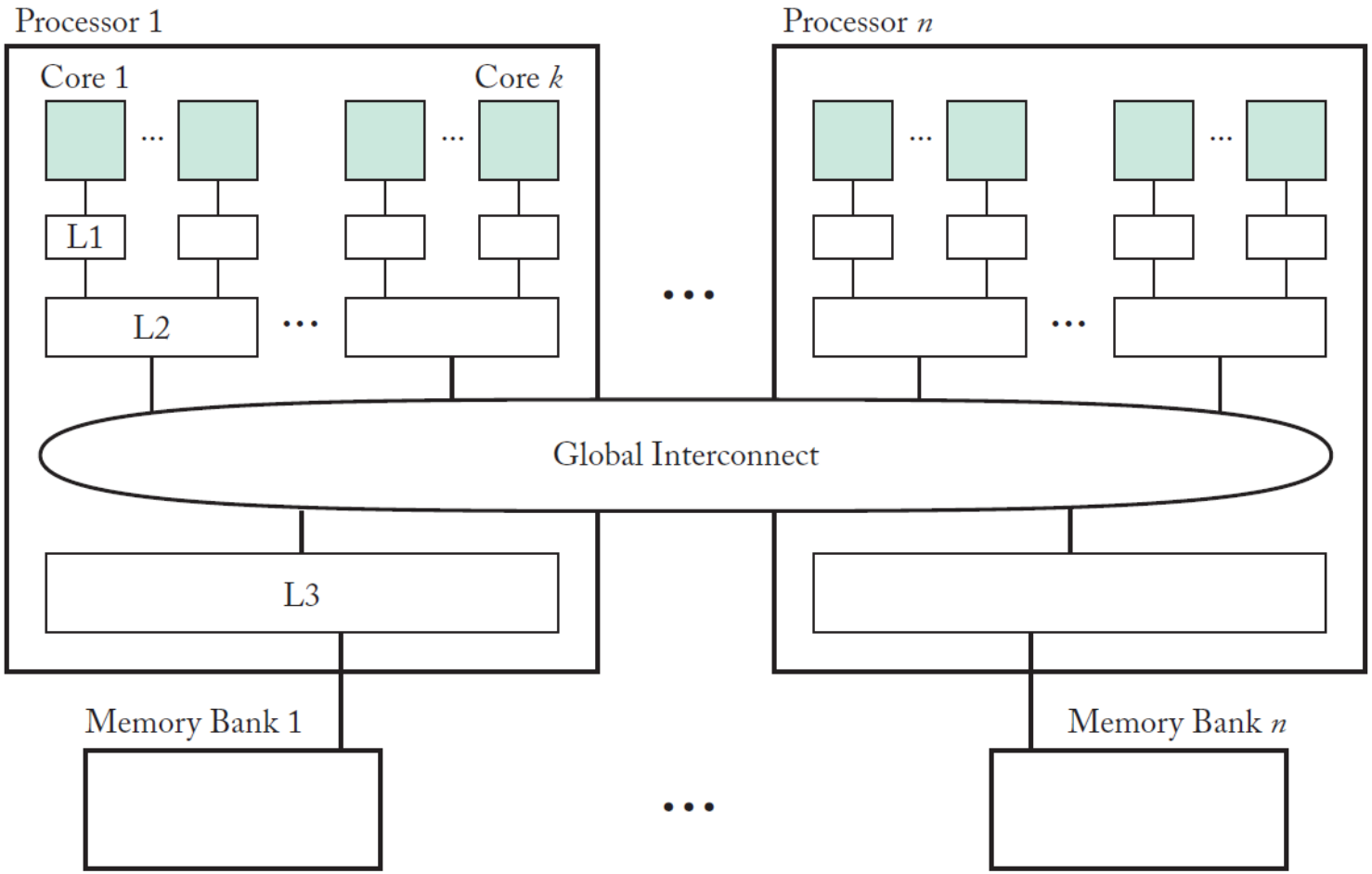
Повторяем материал

«The network is the computer»

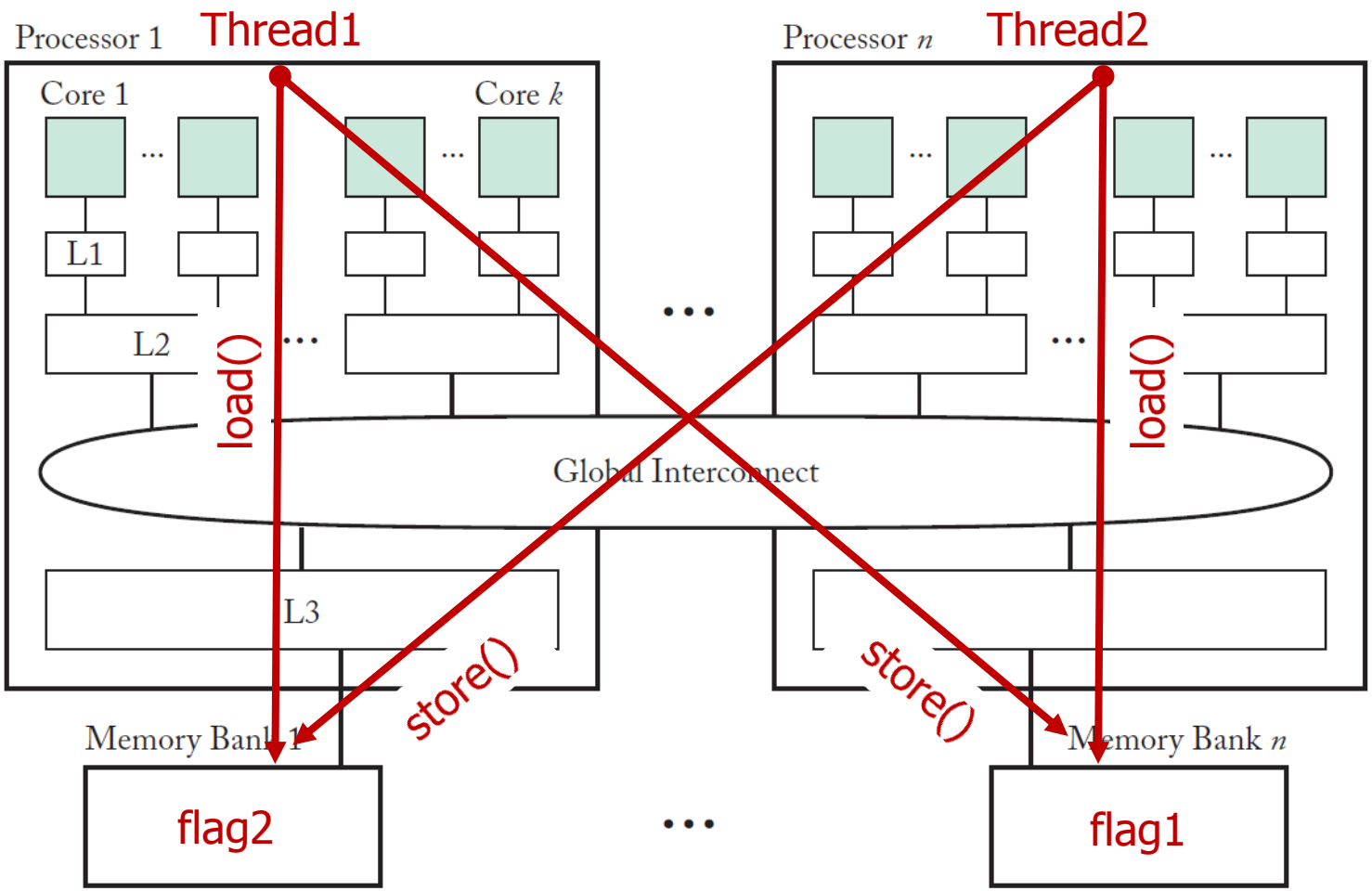
John Gage, Sun Microsystems

«… but the computer is also the network»

Повторяем материал



Повторяем материал



Лирическое отступление

Вопросы (связь теории с практикой):

- У каждого процесса свое смещение (текущая позиция в файле / offset) в лог-файле или у всех общее?
- Операция вывода на терминал / записи в лог-файл атомарна или нет?
- Вывод на терминал / запись в лог-файл буферизована или нет?
- Последовательность вывода записей в лог-файл от всех процессов правильная или нет?
- Используемые примитивы write() записи в канал / read() чтения из канала блокирующие или нет? Отправка / получение буферизована или нет?

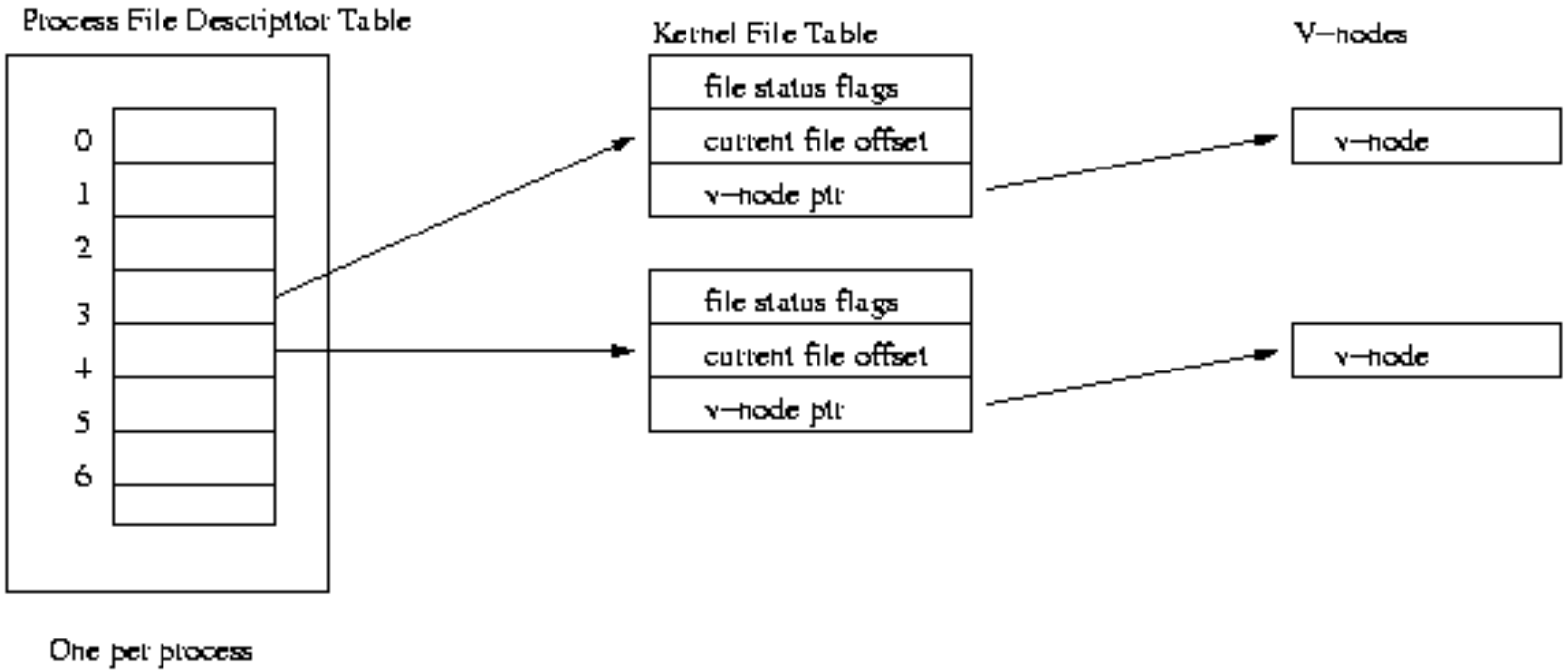
Лирическое отступление
Смещение

Запись в файл:

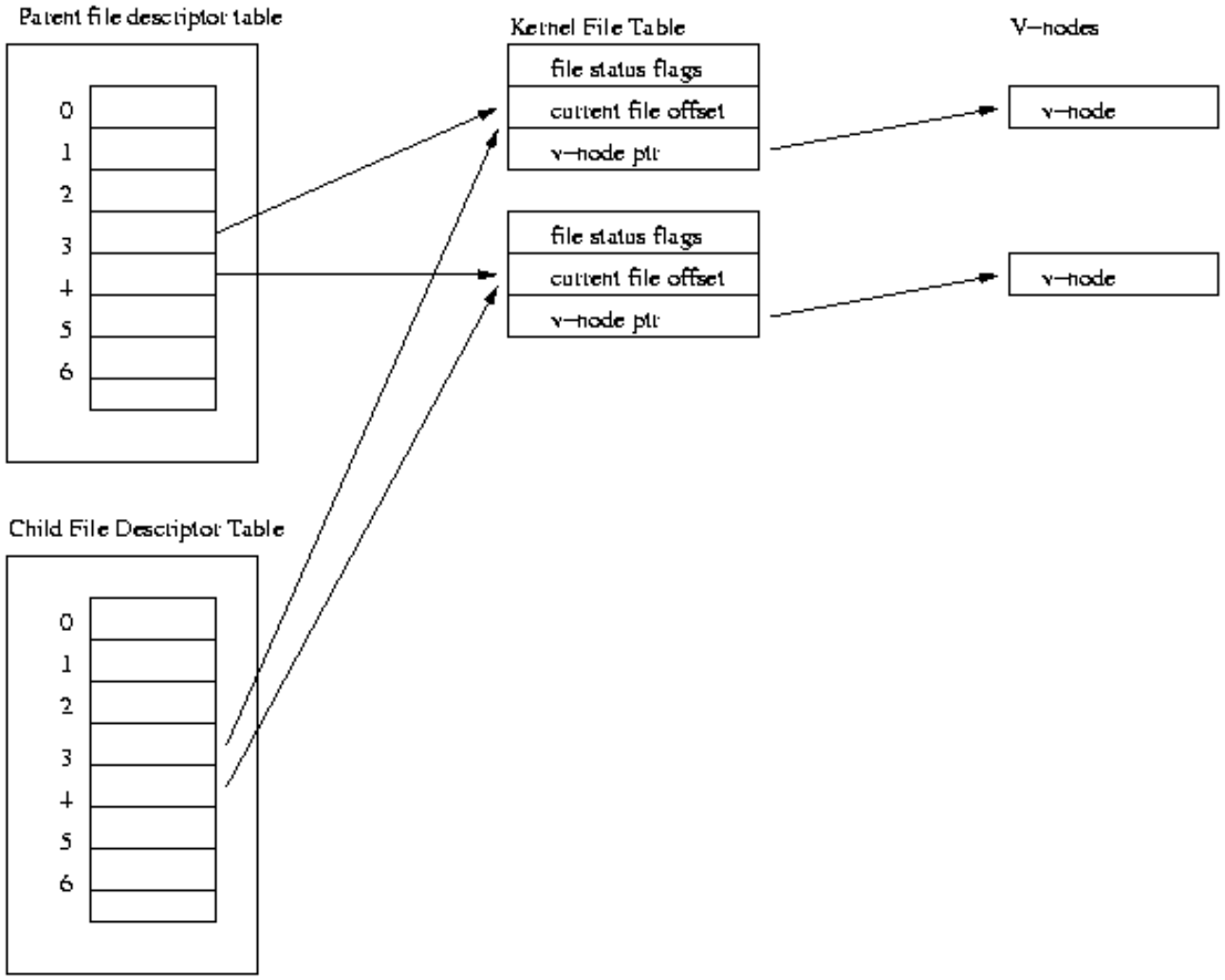
Вариант 1	Вариант 2

fd = open(filename, O_WRONLY | O_TRUNC);
fork();
write(fd, buf, strlen(buf));
	fork();
	fd = open(filename, O_WRONLY | O_TRUNC);
	write(fd, buf, strlen(buf));

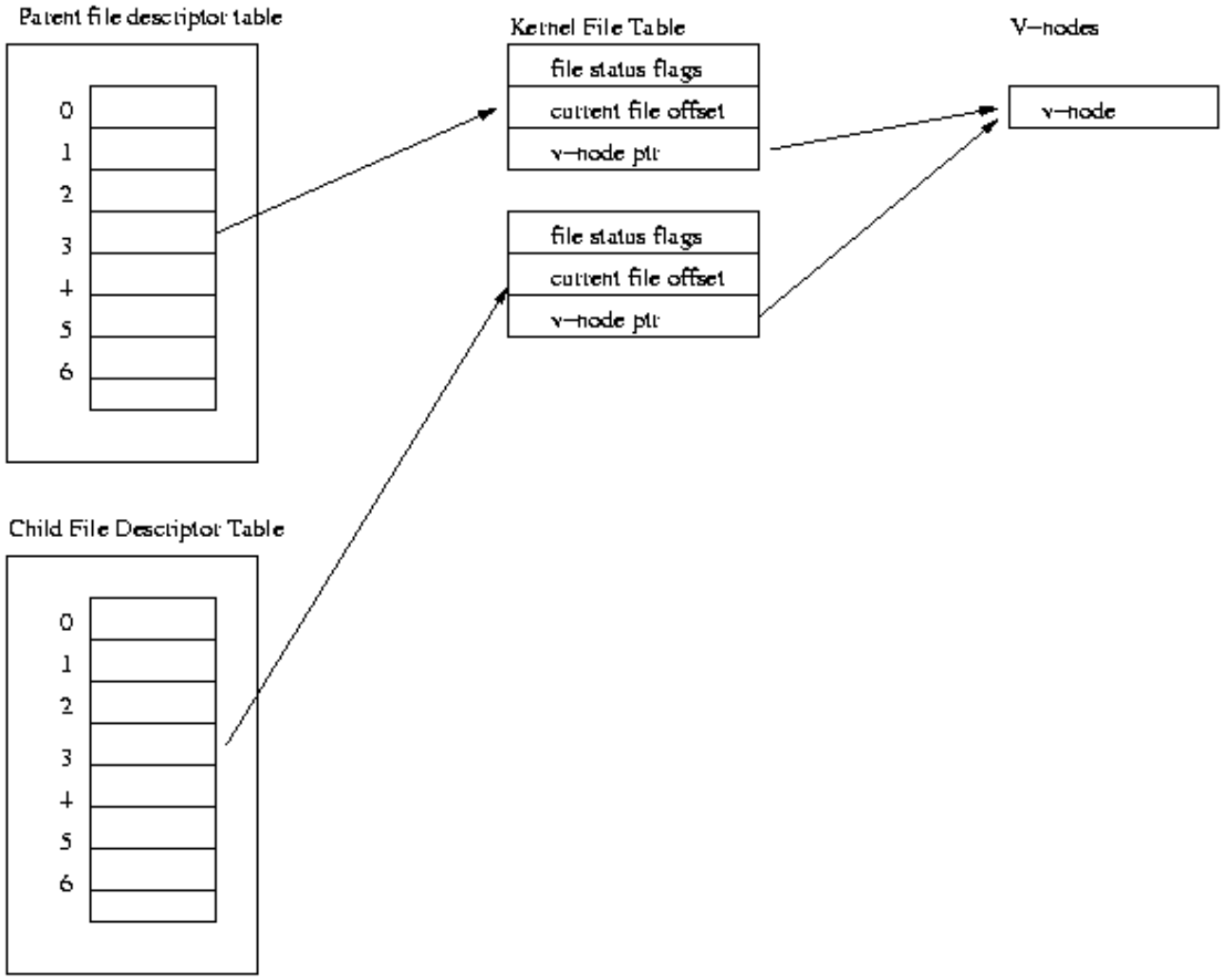
Лирическое отступление
Смещение



Лирическое отступление
Смещение: вариант 1



Лирическое отступление
Смещение: вариант 2



Лирическое отступление
Атомарность

Запись в конец файла:

Вариант 1	Вариант 2

fd = open(filename, O_WRONLY);
fork();
lseek(fd, 0, SEEK_END);
write(fd, buf, strlen(buf));
	fork();
	fd = open(filename, O_WRONLY);
	lseek(fd, 0, SEEK_END);
	write(fd, buf, strlen(buf));

Лирическое отступление
Атомарность

ORC®

Запись в конец файла:

Вариант 1	Вариант 2

fork();
fd = open(filename, O_WRONLY);
lseek(fd, 0, SEEK_END);
write(fd, buf, strlen(buf));
	fork();
	fd = open(filename, O_WRONLY | O_APPEND);
	write(fd, buf, strlen(buf));

If the file was opened with O_APPEND, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

Лирическое отступление
Буферизованная запись

- I/O library functions: fopen, fscanf, fprintf, fclose
- UNIX I/O functions: open, read, write, close

- int open(const char *pathname, int flags);
- FILE *fopen(const char *path, const char *mode);

Вариант 1	Вариант 2

int main(void) {
  printf(“This is my output.”);
  fork();
  return 0;
}
	int main(void) {
	  printf(“This is my output.\n”);
	  fork();
	  return 0;
	}

Лирическое отступление
Буферизованная запись

ORC®

Запись в конец файла:

Вариант 1	Вариант 2

fork();
fd = open(filename,
O_WRONLY | O_APPEND);
write(fd, buf, strlen(buf));
	fork();
	fd = fopen(filename, “a”);
	fprintf(fd, “This is my output”);
	? fflush(fd);
	? setvbuf(fd, NULL, _IONBF, 0);

LET IT TRADE

Лирическое отступление
Ведение лога

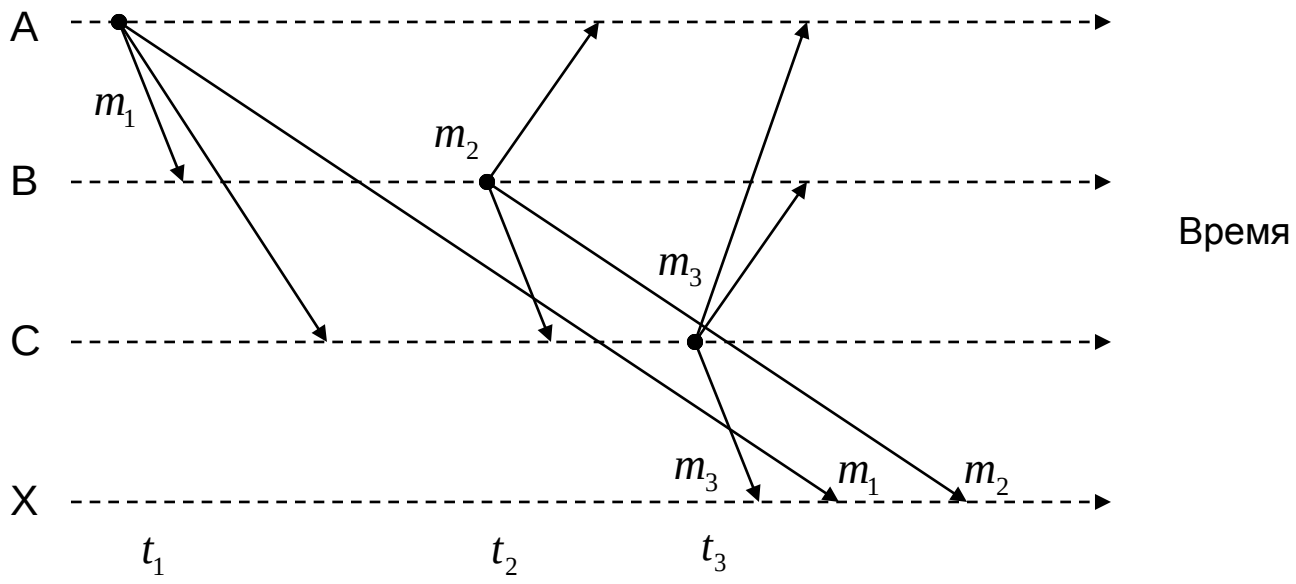
Логирование событий: как обеспечить правильную последовательность записей?

P_1	P_2

send_message();
log_event();
 	
или
 	
log_event();
send_message();
	receive_message();
	log_event();
	
	или
	
	log_event();
	receive_message();

Лирическое отступление
Ведение лога

Мы пытаемся линейно упорядочить события



Сторонний наблюдатель X – либо файл, куда записывается лог, либо процесс, которому отправляются записи о событиях для ведения лога

Примитивы взаимодействия

send(void *sendbuf, int count, int dest)

- sendbuf – указывает на буфер, содержащий передаваемые данные
- count – содержит количество передаваемых элементов данных
- dest – идентифицирует процесс-получатель

receive(void *recvbuf, int count, int source)

- recvbuf – указывает на буфер, сохраняющий принимаемые данные
- count – содержит количество принимаемых элементов данных
- source – идентифицирует процесс-отправитель

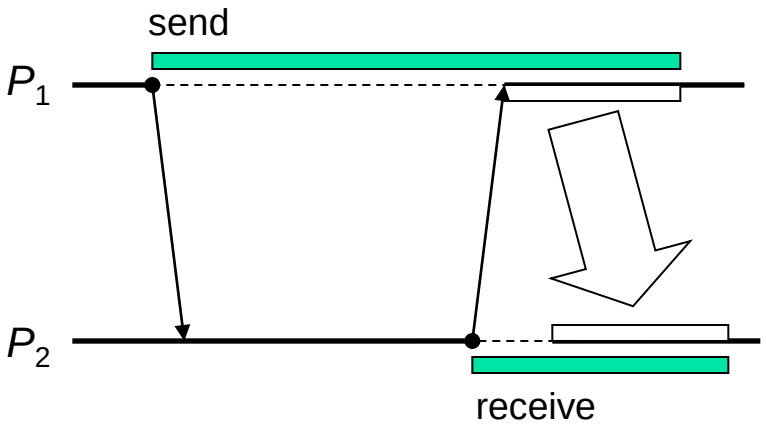
Примитивы взаимодействия

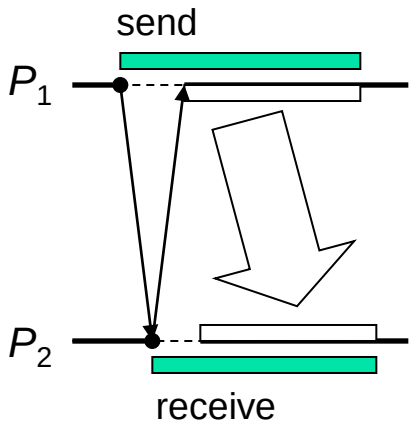
Пример обмена данными:

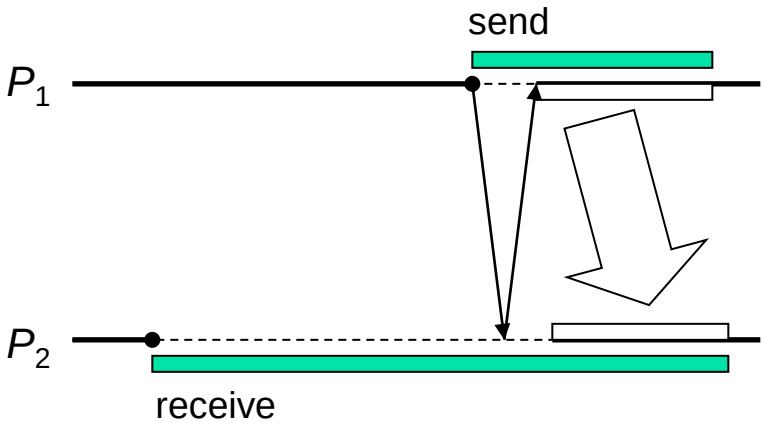
P_1	P_2

a = 100;
send(&a, 1, 2);
a = 0;
	receive(&a, 1, 1);
	printf("%d\n", a);

Блокирующая отправка/ получение без буферизации







Время

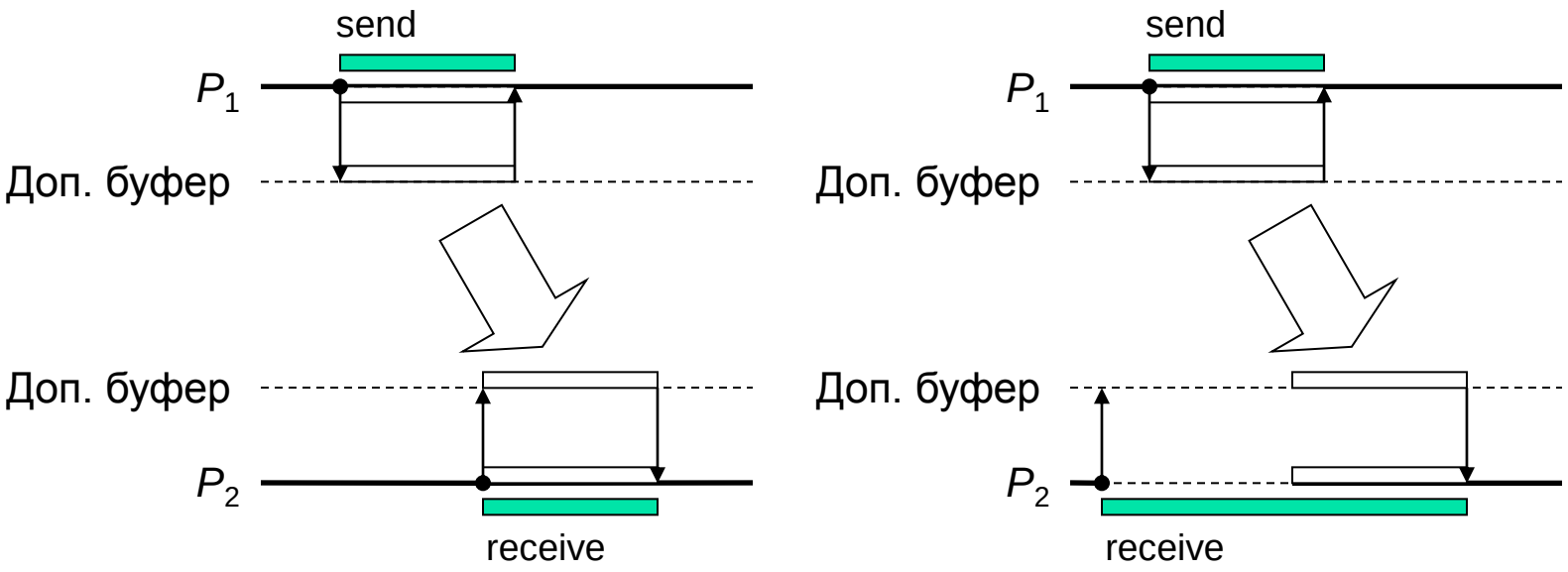
- вызов команды *send()* или *receive()*
- длительность блокировки процесса
- длительность отправки / приема данных

Блокирующая отправка/ получение без буферизации

Пример взаимной блокировки:

P_1	P_2
send(&a, 1, 2); receive(&b, 1, 2);	send(&a, 1, 1); receive(&b, 1, 1);

Блокирующая буферизованная отправка/получение



- вызов команды *send()* или *receive()*
- длительность блокировки процесса
- длительность копирования данных в буфер / из буфера

Блокирующая буферизованная отправка/получение

Пример влияния буферов конечного размера:

P_1	P_2

for (i = 0; i < 1000; i++)
{
produce_data(&a);
send(&a, 1, 2);
}
	for (i = 0; i < 1000; i++)
	{
	receive(&a, 1, 1);
	consume_data(&a);
	}

Блокирующая буферизованная отправка/получение

Пример взаимной блокировки:

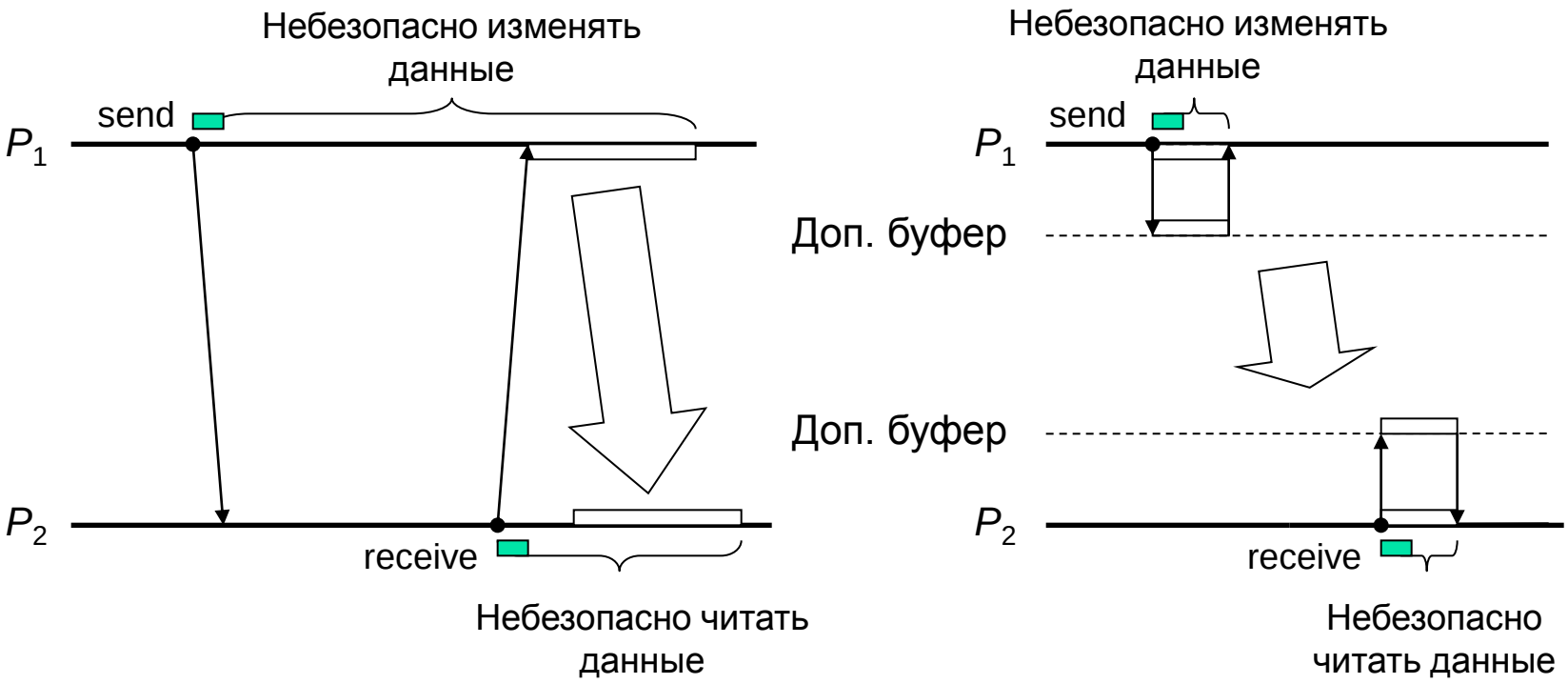
P_1	P_2
receive(&a, 1, 2); send(&b, 1, 2);	receive(&a, 1, 1); send(&b, 1, 1);

Блокирующая буферизованная отправка/получение

И еще раз посмотрим на старый пример:

P_1	P_2
send(&a, 1, 2); receive(&b, 1, 2);	send(&a, 1, 1); receive(&b, 1, 1);

Неблокирующие операции отправки/получения



Синхронный vs асинхронный обмен сообщения

- Блокирующие / неблокирующие примитивы описывают локальное поведение процесса
  - либо возвращают управление вызывающему процессу при завершении копирования данных из(в) адресного(ое) пространства
  - либо лишь начинают операцию
- С глобальной точки зрения интересен синхронный / асинхронный обмен сообщениями

Синхронный обмен сообщениями

- Отправитель и получатель дожидаются друг друга для передачи каждого сообщения
  - операция отправки считается завершенной только после того, как получатель закончит прием сообщения
  - не требует доп. буферов
  - receive() отправляет acknowledgement
  - блокирующие send() + receive() = одна атомарная операция

Асинхронный обмен сообщениями

- Не происходит никакой координации между отправителем и получателем
  - отправителю неизвестно, получено ли его предыдущее сообщение: в т.ч. non-FIFO
  - требует доп. буферов для блокирующих примитивов
  - receive() НЕ отправляет acknowledgement
  - все ради того, чтобы перекрыть вычисления отправителя и получателя во времени

Спасибо за внимание!