處理 signal:

使用 sigaction 函式, 對 SIGPIPE 以及 SIGALRM 都指定自己的函式處理
當碰到 SIGPIPE 時, 印出 "SIGPIPE\n"
每碰到 SIGALRM 5 次時, 印出 "SIGALRM\n"
(因為我設定每三秒一次 SIGALRM, 不然太慢)

```
void useless_sigpipe(int signo) {
    assert(signo == SIGPIPE);
    write(STDERR_FILENO, "SIGPIPE\n", 8);
}

void sigalrm(int signo) {
    static int times = 15/ALRM_PERIOD;
    assert(signo == SIGALRM);
    if (--times == 0) {
        write(STDERR_FILENO, "SIGALRM\n", 8);
        times = 15/ALRM_PERIOD;
    }
    alarm(ALRM_PERIOD);
}
```

因為我不使用 sleep 函數, 所以這樣設 alarm 沒有問題.

對於 pipe, 我使用 epoll 來關注他們.
所以當 epoll 收到 EPOLLIN 時, 使用 read 一定不會 block
收到 EPOLLOUT 時, 則代表 buffer 中有空位可以寫.
當 read pipe 遇到 EPOLLERR 時, 代表 pipe 的讀端都被關掉了,
於是我印出 "SIGPIPE\n" 代表 pipe 壞掉了. 然而, 遇到 EPOLLERR 後,
我並不會真的去讀寫東西, 所以實際上並沒有 signal 產生.
而 pipe 因為在開啟時指定了 O_CLOEXEC,

整個程式的流程大概是:
開好所有的 sorting 程序, 然後分派工作給他們.
對每個工作, 當他可以讀寫時, 就把相對應要讀寫的資料傳給 sorting,
寫是從 stdin 適當的位置讀資料然後寫進去,
讀是從 pipe 讀進 sorting 完成的資料寫到暫存檔去.

因為我的 pipe 並沒有使用 non-blocking mode, 所以寫的時候若超出
buffer 目前空位, 有可能 block 住.

因此, 我們每三秒一次的 SIGALRM 就達到負責把 block 住的 write 喚醒
的作用, 而把剩下沒寫完的部份留到下次做, 才不會卡死.

```
    for (i = 0; i < P; i++) {
        auto it = pending.begin();
        (*it)->attach(procs[i]);
        assoc[i] = *it;
        busy[i] = true;
        pending.erase(it);
        running.insert(assoc[i]);
    }
```

若過程中遇到 sorting 當掉沒回應, 15 秒後, 它會被 kill 掉,
然後重開一份, 再繼續原先的工作.
遇到 pipe 壞掉時, 印出 "SIGPIPE\n", 然後關掉 sorting 再重開.

```
    while (!pending.empty() || !running.empty()) {
        logger.print("wait");
        Watcher::getInstance().wait();
        for (i = 0; i < P; i++) {
            if (!busy[i])
                continue;
            if (!procs[i]->responsed()) {
                logger.print("process %p has no response", procs[i]);
                assoc[i]->detach();
                delete procs[i];
                procs[i] = new Subprocess(sorting.c_str());
                assoc[i]->attach(procs[i]);
            }
        }
    }
```

這過程中沒看到讀寫, 是因為 "有空/有資料可以讀寫" 是透過事件
的方式傳遞, 會在 Watcher 的 wait() 中傳送事件到正確的物件
然後該物件 (e.g. Task) 會負責讀寫.

當所有的工作都排序完成後, 用個 heap 把它 merge 起來,
因為 heap 可以很方便取得最小的數字. 然後直到第 k 個數字前, 都丟掉,
最後輸出第 k 個數字.

```
    int topidx, heap_c = N, heap[NMAX];
    int32_t num[NMAX];
    auto cmp = [&num](int idxa, int idxb) { return num[idxa] > num[idxb]; };
    for (i = 0; i < N; i++) {
        num[i] = tasks[i]->getInteger();
        heap[i] = i;
    }
    std::make_heap(heap, heap + heap_c, cmp);
    while (--K) {
        topidx = heap[0];
        std::pop_heap(heap, heap + heap_c, cmp);
        if (tasks[topidx]->eof()) {
            heap_c--;
            continue;
        }
        num[topidx] = tasks[topidx]->getInteger();
        std::push_heap(heap, heap + heap_c, cmp);
    }
    printf("%d\n", num[heap[0]]);
```

整體而言, 這份 code 的架構如下：

watcher.h watcher.cpp	使用 epoll 來監聽 fd 的 Singleton, 並 deliver events
pipe.h pipe.cpp	把 pipe 包裝成一個物件
subproc.h subproc.cpp	fork執行子程序的物件 並把 stdin, stdout 導向到 pipe
task.h task.cpp	把一段 sorting 工作包裝成一個物件 並處理輸出入
log.h log.cpp	印出 log 以及 stack trace 的物件, 本次沒使用
mrg.cpp	整個排序的工作, Sched
srt.cpp	sorting

```
/* watcher.h */

class Listener { /* event listener 的介面 */
```

```
    public:
        virtual void action(Listener*, int) = 0;
        virtual ~Listener();
};

class Watcher {
    private:
        int epollfd;
        std::unordered_map<int, Listener*> watching;
    public:
        static Watcher& getInstance();

        void watch(int, Listener*);
        void unwatch(int);
        void wait(); /* 等待事件並 deliver */
        Watcher();
        ~Watcher();

        Watcher(const Watcher&);
        const Watcher& operator=(const Watcher&);
};



/* pipe.h */

class Pipe { /* 一個 pipe */
    protected:
        int rd, wr;
    public:
        Pipe();
        ~Pipe();
        int useWriteEnd();
        int useReadEnd();

        Pipe(const Pipe&);
        const Pipe& operator=(const Pipe&);
};

class PipeEnd : public Listener { /* pipe 的一端 */
    protected:
        int pipefd;
        Listener *listener;
    public:
        void action(Listener*, int);

        void dupTo(int);
        void setListener(Listener*);
        PipeEnd();
        ~PipeEnd();

        PipeEnd(const PipeEnd&);
        const PipeEnd& operator=(const PipeEnd&);
};

class WriterPipe : public PipeEnd { /*  pipe 的寫端 */
    public:                         /* 並關掉不要的 fd */
```

```
        WriterPipe(Pipe&);
        ssize_t write(size_t, void*);

        WriterPipe(const WriterPipe&);
        const WriterPipe& operator=(const WriterPipe&);
};

class ReaderPipe : public PipeEnd { /*  pipe 的讀端 */
    public:                         /* 並關掉不要的 fd */
        ReaderPipe(Pipe&);
        ssize_t read(size_t, void*);

        ReaderPipe(const ReaderPipe&);
        const ReaderPipe& operator=(const ReaderPipe&);
};




/* subproc.h */

class Subprocess : public Listener {
    private:                    /* fork , 開子程序 */
        pid_t pid;
        ReaderPipe *rd;
        WriterPipe *wr;
        time_t lastResponse;

        Listener *listener;
    public:
        void action(Listener*, int);
        void poke();
        bool responsed();

        Subprocess(const char*);
        ~Subprocess();
        void setListener(Listener*);
        size_t readFrom(size_t, void*);
        size_t writeTo(size_t, void*);

        Subprocess(const Subprocess&);
};



/* task.h */

class Task : public Listener { /* 排序工作分段 */
    private:
        int begin, end; // index
        FILE *ftmp;
        Subprocess *proc;
        Listener *listener;

        int bytesRead, bytesWritten;
        char buffer[65536];
    public:
```

```
        void action(Listener*, int);

        Task(int, int);
        ~Task();
        void attach(Subprocess*); /* 分派給某個 sorting */
        void detach();
        bool finished();
        void setListener(Listener*);
        void reset();             /* rewind tmp file */
        int32_t getInteger();
        bool eof();

        Task(const Task&);
        const Task& operator=(const Task&);
};




/* mrg.cpp */

class Sched : public Listener { /* 排序工作排程分配 */
    private:                    /* 以及 merge 的工作 */
        int P, N, K;
        string sorting;
        unordered_set<Task*> pending, running;
        Task *tasks[NMAX], *assoc[PMAX];
        Subprocess *procs[PMAX];
        bool busy[PMAX];
    public:
        void action(Listener*, int);
        Sched(int, char **);
        ~Sched();
        int run();
};
```